1999-09

Modeling control channel dynamics of the SAAM Architecture using the NS network simulation tool

Tiefert, Brian E.

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

MODELING CONTROL CHANNEL DYNAMICS OF THE SAAM ARCHITECTURE USING THE NS NETWORK SIMULATION TOOL

by

Brian E. Tiefert

September 1999

Thesis Advisor: Geoffrey Xie

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1999.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Modeling Control Channel Dynamics of the SAAM Architecture Using the NS Network Simulation Tool

5. FUNDING NUMBERS

6. AUTHOR(S) Brian E. Tiefert

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(maximum 200 words)*

The explosive growth of the Internet and the advent of real-time network applications have stretched the capacity of current network technology. It has become evident that to realize the full potential of the Information Super Highway a new network architecture would have to be developed. It was for these reasons the Next Generation Internet Project was started. As a part of this effort the Server and Agent based Active network Management (SAAM) Project was started. SAAM is a server based hierarchical routing architecture designed to provide Quality of Service (QoS) routing services for network resource intensive applications. Because the study of this topic entailed emulating large Wide Area Networks, a simulation of the entire architecture would have to be developed. This thesis provides the first step towards achieving that goal. The model developed as the basis for this thesis concentrates on the control traffic overhead required to configure and implement the routing mechanism of SAAM. Specifically it simulates the control channel dynamics required to pass control messages between servers, routers and real-time applications.

14. SUBJECT TERMS SAAM, Quality of Service, Network Simulation

15. NUMBER OF PAGES 132

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

MODELING CONTROL CHANNEL DYNAMICS OF THE SAAM ARCHITECTURE USING THE NS NETWORK SIMULATION TOOL

Brian E. Tiefert
Major, United States Marine Corps
B.S., University of Florida, 1988
M.S.B.A., Boston University, 1993

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: [redacted]

Brian E. Tiefert

Approved by: [redacted]

Geoffrey Xie, Thesis Advisor

[redacted]

Michael J. Holden, Second Reader

[redacted]

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

The explosive growth of the Internet and the advent of real-time network applications have stretched the capacity of current network technology. It has become evident that to realize the full potential of the Information Super Highway a new network architecture would have to be developed. It was for these reasons the Next Generation Internet Project was started. As a part of this effort the Server and Agent based Active network Management (SAAM) Project was started. SAAM is a server based hierarchical routing architecture designed to provide Quality of Service (QoS) routing services for network resource intensive applications. Because the study of this topic entailed emulating large Wide Area Networks, a simulation of the entire architecture would have to be developed. This thesis provides the first step towards achieving that goal. The model developed as the basis for this thesis concentrates on the control traffic overhead required to configure and implement the routing mechanism of SAAM. Specifically it simulates the control channel dynamics required to pass control messages between servers, routers and real-time applications.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Computer networking technology has experienced explosive growth over the past decade. However, increases in volume of network traffic are rapidly outgrowing the ability of existing networks to provide the adequate Quality of Service (QoS) they require to function properly. The Next Generation Internet (NGI) Initiative was developed in order to solve this problem. In a concept paper describing the NGI Initiative the goal was stated as follows:

> Today's Internet suffers from its own success. Technology designed for a network of thousands is laboring to serve millions. Fortunately, scientists and engineers believe that new technologies, protocols, and standards can be developed to meet tomorrow's demands. These advances will start to put us on track to a next generation Internet offering reliable, affordable, secure information delivery at rates thousands of times faster than today. [NATI97]

One proposal developed to realize the goals of the NGI Initiative was The SAAM Project. SAAM stands for Server and Agent based Active network Management. SAAM is a DARPA funded project initiated by Dr. Geoffrey Xie at the Naval Postgraduate School in Monterey, California. SAAM is intended to define an architecture to achieve the goals of the NGI Initiative. The concept of the SAAM Project is to use a server based hierarchical architecture to compute and disseminate routing information in order to support guaranteed Quality of Service (QoS) network traffic. This type of QoS guarantee is required in order to assure satisfactory performance of many emerging real-time

network applications, such as voice, video and distributed computing. It is these types of applications that are driving the requirement for the Next Generation Internet.

B. MOTIVATION

SAAM is still under development and many of the underlying concepts require study and refinement. Because SAAM is being developed to be implemented over very large networks, simulation is the only scaleable method to accomplish proof of concept type studies. No hardware has been developed, as of yet, to implement the SAAM Architecture. Emulation software for SAAM routers and servers that operate at the application layer is under development. It is important that a simulation model be created that will interact with these emulation applications to broaden the SAAM research.

In order to create a suitable SAAM simulation model, a simulation software product either had to be developed or an existing one adapted to meet the requirements presented by the architecture. Because of the complexity and difficulty associated with developing simulation tools it was deemed that adaptation of an existing simulation package was the appropriate choice. The simulation package had to be able to be modified to allow for development of new routing protocols and message passing schemes. In addition, the simulation had to be able to interface with the emulation software being developed for the SAAM routers and server. This meant interjecting real network traffic into the simulation for the purpose of stressing the newly developed emulation products.

C. PURPOSE

The purpose of this thesis is to select an existing network simulation tool and develop a preliminary model of the SAAM Architecture. Specifically this model will concentrate on the message passing schemes between the SAAM servers, routers and the applications which will operate over the entire architecture. The purpose of the messages being transmitted is to relay control information between the entities of the architecture in order to coordinate the appropriate routing information in a timely and scaleable fashion.

These "control messages" will carry routing information, network state information, flow requests and flow responses. In the SAAM Architecture the SAAM server is responsible for making all routing decisions for all QoS flows. In order to accomplish this the server is required to maintain as accurate a picture of the network-state as possible. It maintains this view of the network-state by receiving updates from the routers in its region. By maintaining this picture it can accurately calculate the most advantageous path to route flows as requested by applications requiring certain levels of QoS. The model developed for this thesis provides an adequate message-passing scheme to accomplish these tasks.

The simulation package chosen for this thesis was the NS network simulation tool. NS is a product of The University of California, Berkley. It was developed as part of the VINT Project. It is an open source network simulation tool and it is a stochastic discrete event simulator. Because the source code is available it is modifiable and configurable. NS has been under development in excess of 10 years and is fairly robust. It contains all of the prerequisites needed to accomplish the objectives of this thesis.

D. EXECUTIVE SUMMARY

In order to build an appropriate model of the SAAM Architecture an extensive analysis of the NS Network Simulator was conducted and the intricacies of its operation examined. Once this analysis was completed it had to be determined what modifications had to be made to the simulator so that it could model the SAAM Architecture. As will be discussed later, the simulator models network behavior by utilizing nodes as network entities. On these nodes agents can be attached to simulate different network protocols. Scripts are used to schedule network events. To model the SAAM Architecture three new agents had to be developed. Two of these agents were to emulate new types of network entity, a SAAM router and a SAAM server. The third agent to be developed was a special type of agent that is used to model network routing protocols.

The SAAM Server agent was required to emulate the actions that take place in a notional SAAM server. The details of the SAAM Architecture will be discussed in detail, but in general the SAAM server has to be able to accomplish several tasks. First the server has to be able to maintain a Path Information Base (PIB). The PIB is a detailed table of the entire network that the server is responsible for controlling including all links between nodes and the pertinent QoS parameters inherent to each link. The server must then be able to accept messages from applications requesting flow assignments that require a certain level of quality of service that are defined by expressing them in terms of the set parameters.

Once these flow requests are received the server must be able to calculate an appropriate path from source to destination that meets the quality of service requirements requested from the application. Once the path is determined for a flow the server must

send flow routing table updates to each router that is associated with each link in the determined path. Once all of the associated routers have been updated the server then must be able to send a flow response to the requesting application.

The SAAM Router agent had to be able to send messages to the server in order to keep the server informed of the state of all of its associated links in terms of the quality of service parameters. In addition it had to be able to update a flow routing table that resides within the protocol agent. The flow routing table is used to forward packets assigned to a flow to the designated next hop in the network. In order to maintain the flow routing table the router had to be able to receive flow routing table updates from the server.

The protocol agent is the entity in the simulator that actually forwards the packets. As mentioned above the protocol agent actually maintains the flow routing table. The protocol agent uses the flow routing table to lookup next hop information based on a flow identification fixed to each packet by the source application as assigned by the server. Once the next hop has been determined the protocol agent actually forwards the packet to the next hop. This procedure repeats at each node in the assigned path until the packet reaches its destination.

E. BENEFITS OF STUDY

With the advent of new technologies and the requirement to support near real-time network traffic, a routing architecture that can schedule network resources and adapt to changing environments is required. SAAM provides such an architecture. In order for SAAM to make appropriate routing decisions it must be able to track and adjust to volatile network conditions. Information concerning these conditions must be passed

from router to server and back rapidly while consuming a minimum of network resources. This study will provide, if not a method for doing that, at least a better understanding of the problems faced in the attempt. The Next Generation Internet and QoS dependent applications will become a reality and will be in common use in the near future. This research and the lessons learned from the SAAM Project will provide a framework for capitalizing on these technologies.

The model developed in this thesis provides a platform from which future research can be conducted into the SAAM concept. Comparisons of control message passing schemes, routing protocols, fault tolerance implementations and new transport protocols can all be made through use of this model with minimal modification. The ability of researchers to test and compare and develop these concepts in a SAAM context will not only extend progress in the SAAM Project, but can also further concepts required for NGI research in general. Many of these concepts can be adapted to be incorporated into new or existing network architectures that support the type of QoS routing that is required to implement NGI goals. The ability to develop and test these concepts is essential to progression of a coherent NGI model, and can significantly reduce development time required to make NGI a reality.

The SAAM project is intended to provide a mechanism for managing integrated services and providing real-time performance guarantees. If successful, the benefit to the Department of Defense (DoD) would be tremendous. The impact of SAAM's capabilities would be far-reaching and could revolutionize the way DoD organizations are able to operate and communicate. It would significantly enhance the department's ability to disseminate information on and off the battlefield. Such capabilities could affect areas

ranging from planning, warfighting, command and control and simulation. The ability to transmit and receive real-time information and data is essential to information-centric warfare.

F. ORGANIZATION

The remainder of this thesis is organized as follows:

1. Chapter II

Chapter II is a detailed description of the SAAM Architecture with special emphasis on the control channel message passing apparatus. The chapter will discuss SAAM in theory and outline proposed topologies for implementation. It will also discuss new network hardware that will be required to implement SAAM in the real world. An overall illustration of interactions between network entities in the SAAM Architecture is also included. In addition, it provides details of the types and structure of the proposed control messages required to implement the SAAM Architecture. Finally, a comparison to existing proposals for QoS routing is made and discussion of how portions of these proposals may be incorporated into SAAM.

2. Chapter III

Chapter III is an analysis and description of the NS Network Simulation Tool. The reasons why this tool was chosen are discussed along with the internal architecture of the tool. The analysis includes the duality of objects in both C++ code and Tcl code as

utilized in the NS architecture. Descriptions of required classes in both programming languages are provided along with detailed analysis of the interactions between these classes during simulations.

3. Chapter IV

Chapter IV is a detailed analysis of the model built. It includes a discussion of all developed and modified classes and their purposes. Descriptions of each of these classes are given along with their interactions with other NS classes. Detailed explanations of how these classes provide realizable implementations of the SAAM Architecture are provided.

4. Chapter V

Chapter V is a summary detailing strengths and weaknesses of the model and a discussion of topics for follow on research and development. The chapter is meant to illustrate what portions of the SAAM Architecture were modeled and the relative successes or failures that they represent. In addition, it provides a review of the NS simulation tool and its appropriateness for this study. Finally, the chapter provides recommendations for extended use of the model for further research and proposed modifications to enhance the capabilities of other researchers.

II. SAAM CONCEPT

A. OVERVIEW

SAAM is a unique concept developed in order to realize the goals of the NGI Project. Specifically its goals are to provide an architecture that will provide for guaranteed quality of service for applications that require certain levels of network performance for optimal functionality. To maintain this level of quality of service the architecture must be able to provide a number of distinct characteristics. These characteristics include responsiveness, fault tolerance, adaptability, and scalability. As will be explained, SAAM has provisions to address all of these criteria.

The SAAM concept is a departure from the router-based architecture prevalent throughout the Internet today. The fundamental precept is to use a hierarchy of SAAM servers to control and guide a network of light-weight routers in order to maintain a guaranteed QoS for applications that require a certain level of performance. This level of performance can be measured by several means. The most prevalent and important metrics include packet delay, packet loss and throughput.

B. ARCHITECTURE

The architecture is based on a hierarchy of logical servers and lightweight routers. Each SAAM server is responsible for collecting and aggregating routing and topology information for its subsection of the network, or autonomous region. The server aggregates this information and advertises it to the next higher level in the hierarchy. An

example hierarchy is shown in Figure 1. This mechanism provides the architecture scalability by creating an exponential reduction in the passing of duplicate information between routers and servers.

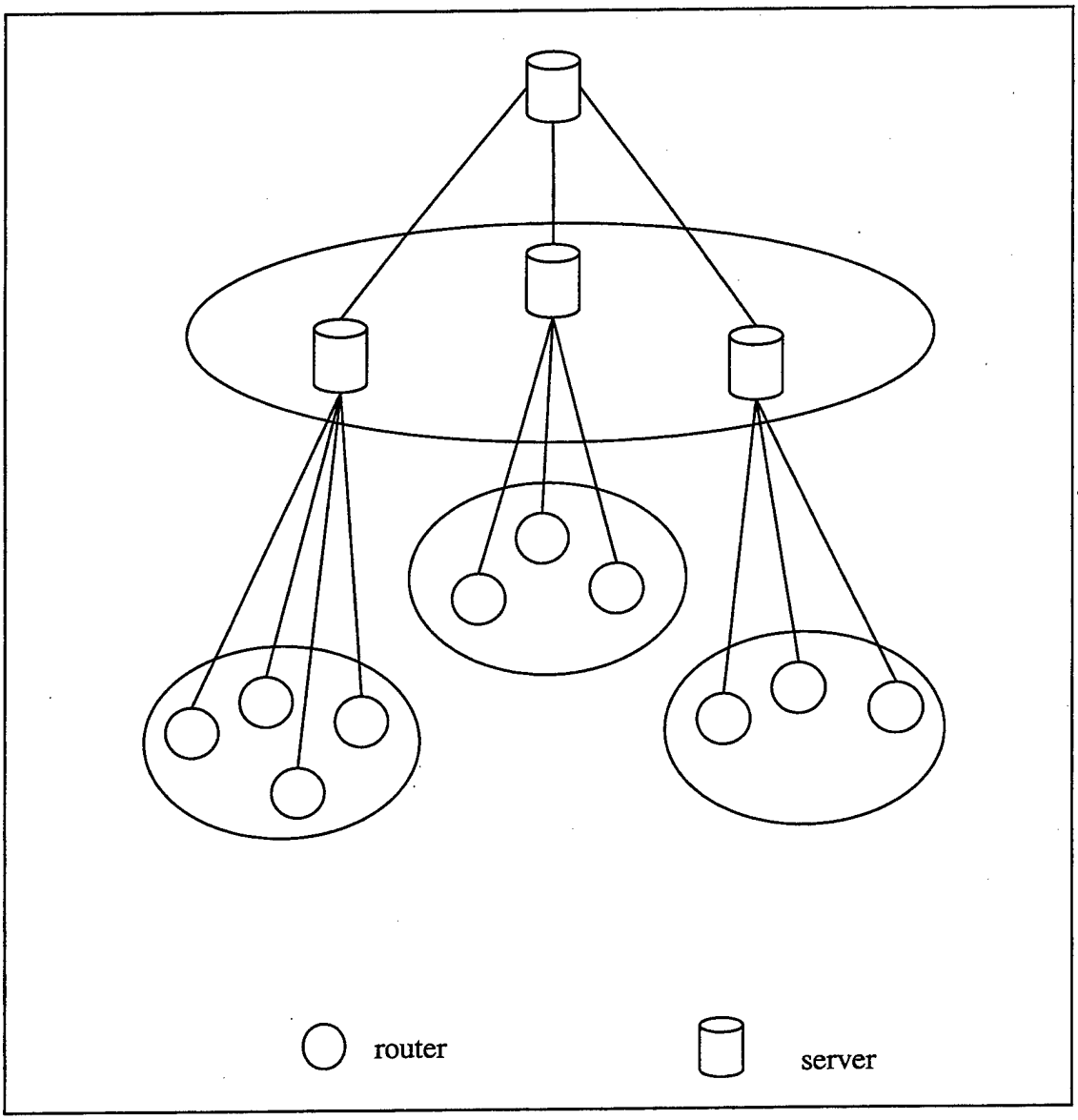


Figure 1. Example SAAM Hierarchy

Each server collects the topology and routing information for its region from the subordinate routers and servers in that region. This information is maintained in a Path

Information Base (PIB) at the server. The PIB includes every path from each source to each destination in the region. The path consists of not only the logical route a packet would travel from source to destination, but also the specific QoS parameters available on that path. The server's PIB forms the basis from which all routing decisions within the region are made.

An application requiring a minimum level of performance would make a request from the server to register its data flow. All routing decisions for QoS flows are made at the server. Once the server receives the request it calculates a path that would satisfy the application's QoS requirement by consulting its PIB. Once an adequate path is identified all routers in that path have to be notified and updates made to their routing tables. All routing table entries for QoS flows at the router are received from the server. By calculating routes at the server, the routers have minimal processing required to forward packets, as each packet belonging to a flow will travel along the same path.

The routers are responsible for maintaining statistics on the state of all of their associated links so that they can advertise this information to the server. It is these Link State Advertisements (LSA's) that the server uses to build and update its PIB. The more frequently the routers send LSA's to the server the more accurate a picture of the state of the network the server can maintain. However, LSA's travel in band, that is they are sent along the same physical medium as all other traffic. This can present a problem as a link becomes more congested. SAAM presents mechanisms for balancing the requirement for LSA's and other control channel information with the need for making the maximum amount of network resources available for normal traffic. Also, there is a need for a

mechanism to ensure that LSA's reach the server. This is the purpose of the control channel.

For efficiency the SAAM architecture needs to be able to rapidly adapt to volatile changes in network traffic patterns and link states. To achieve this rapid response SAAM will employ two methods. The first method is the triggering of link state advertisements by significant change of link state. The second method entails the use of active agents deployed by the server to poll information from various points in the network to assess the efficiency of the current configuration and routing patterns. Information provided by these active agents will be utilized to update the PIB in order to make better routing decisions.

An advanced concept proposed to be incorporated in future versions of SAAM is the use of active agents in conjunction with a simulation protocol in the attempt to predict future network traffic patterns. Such a prediction would allow the SAAM structure to prepare for additional traffic and perform load balancing so that a greater efficiency can be achieved and a higher level of fault tolerance can be realized. To date this concept has not been successfully implemented in any of the currently used routing architectures.

C. QOS PATH MANAGEMENT

One of the major challenges in implementing SAAM is QoS path management. The concept to be employed by SAAM partitions each physical link into service level pipes as shown in Figure 2. Each of these service level pipes is defined by a set of QoS parameters. [XIE98] These parameters include a bound on packet delay, rate of packet

loss and available bandwidth. An application requiring a certain level of QoS will make a request from the SAAM server. The server will determine a suitable path for the application's data flow by consulting its PIB. This path is composed of a series of service level pipes whose composite QoS metrics meet the specified requirements. [XIE 98]

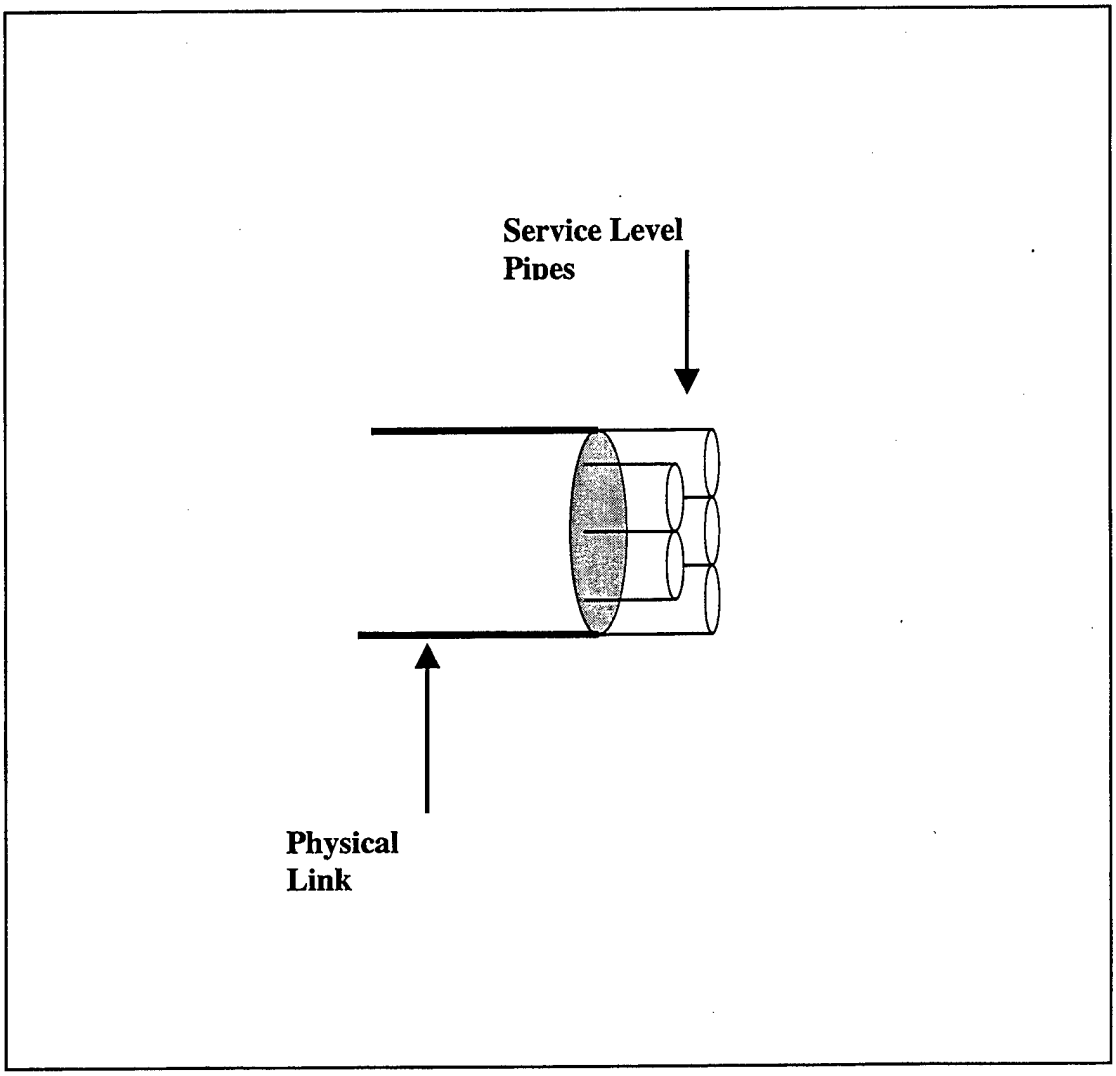


Figure 2. Partitioning of a Physical Link

Path management not only includes the initial selection and assignment of a path to a flow, but also the monitoring and possible reassignment of paths to flows based on

changing network dynamics. If there is a degradation of performance in any of the QoS parameters associated with a path that violates the source's original requirement, then a reconfiguration is required which will necessitate that new paths be assigned to affected flows. This process must be accomplished automatically, before the user perceives a degradation of service. This level of fault tolerance will be revolutionary by today's standards.

Each service level pipe will be associated with an outbound packet queue above each network interface. The QoS parameters, which define each service level pipe, will be maintained partly through queue management. Packet loss and delay statistics can be controlled by varying queue size and packet scheduling. By manipulating packet flow through various queues SAAM will be able to dictate the QoS deliverable across all service level pipes. This is more advantageous than most reservation protocols, which earmark network resources, for particular flows whether they utilize the resource or not. The SAAM protocol allows network resources to remain available for use by any traffic, but ensures that those with QoS guarantees are serviced within acceptable limits.

D. CONTROL CHANNEL

Management of the SAAM system requires communication of control information such as LSA's, routing table updates, and flow requests and responses. In order to maintain an efficient configuration, the SAAM architecture requires timely delivery of this control information. The mechanism for delivering control messages is the control channel. The control channel is a logical partition of the physical route that control

messages are transmitted on. It can be viewed as a distinct service level pipe. Because control messages are perishable and timely delivery is important, the control channel must have the highest priority of all the service level pipes.

In order for the server to make appropriate and timely routing and configuration decisions, it must receive and maintain the most current available information pertaining to network state down to the individual link level. It receives this information in by two means, probing and Link State Advertisements (LSA's). Probing is the use of active agents by the server to dynamically poll usage and latency information from subsections of the network. LSA's are the chief method used by the routers to update the server's network awareness. Both of these methods require use of the control channel to transmit information to and from the server.

LSA's are the principal method for transmitting network state information to the server and will typically make up the majority of control traffic flowing through the control channel. Minimizing the number of LSA's required to update the server is a major objective of SAAM. Therefore, determining when to send an LSA becomes an important question. There are two possible methods of triggering LSA's. The first is by timed intervals. This method entails the router sending an LSA at regular intervals whether or not any change in link-state has occurred. Utilizing this method allows very regular patterns of topology updates so that the server can maintain a view of the network with a fidelity dependent upon the stability of the traffic patterns.

The disadvantage of using timed intervals between LSA's is that modern network traffic is extremely bursty. To minimize the amount of control traffic, the interval between LSA's would be increased. However, with this increased interval major changes

in link-state could occur which should affect routing decisions, but because they occurred between LSA's the server will be unaware of them until the next update. In the interim several poor routing decisions could be made due to expired information at the server. To improve this the interval could be decreased to minimize the chances of a link-state change not reaching the server in time. This, however, would increase the overall control traffic, which we are trying to minimize. In addition it would be inherently wasteful, as many LSA's would be sent that would reflect no substantive changes in the server topology information. Advertised changes may be of a short duration and subsequent advertisements may just cancel out.

The second method is to use a trigger. The trigger is a measurement in vital link-state statistics. If the rate of change in any QoS parameter crosses its identified threshold then a link-state advertisement is constructed and sent to inform the server. In this way only meaningful changes in link-state would be advertised. This method also presents problems. If a link or router were to go down or be taken out of service then the server would not receive an LSA from it. The server might mistakenly interpret the lack of information as meaning that no change in its link-state has occurred and continue to route traffic over the downed link.

The solution to this dilemma is to combine the two methodologies. The first step is to incorporate a trigger scheme much like that presented in [GUER98]. The idea behind this philosophy is to set up thresholds in rate of change in link-state parameters to trigger an LSA, but to use a hold-down timer to delay the sending of the LSA. This ensures that every significant change in link-state is advertised unless it is negated by a subsequent change that occurs before the hold-down timer expires. The length of the

interval for the hold-down timer can be used as an adjustment to fine-tune the system. It can be adjusted to compensate for the volatility and "burstiness" of the network traffic.

The second step is to send periodic LSA's at less frequent intervals. The purpose of the periodic intervals is to ensure confirmation that a link is operational and update link-state status even when no change large enough to trigger an LSA by the first method has occurred. The interval between periodic updates starts from the time the last LSA was sent regardless of which method triggered the transmission. By using this methodology the network manager can adjust to the dynamics of the traffic. Meanwhile the SAAM system as a whole has been able to minimize control traffic while providing timely information to the server from which it can make appropriate routing and configuration decisions.

THIS PAGE INTENTIONALLY LEFT BLANK

III. NS

A. HISTORY

The NS simulator began as a variant of the REAL (REalistic And Large) network simulator in 1989. REAL was primarily developed to evaluate queuing theory and algorithms in networks. [KESH] NS is now an integral element of the VINT (Virtual InterNetwork Testbed) Project. The goal of VINT is to transform network protocol design and engineering practices in the same way that simulation and VHDL-based methods transformed chip and board level design. NS provides the simulation platform for the VINT Project and is intended to provide a framework for composing simulation modules that will create synergy between disjoint simulation efforts and enable the simulation of complex interdependencies between protocols. [SCHO96]

NS Version 1 was a simulation tool developed by the Network Research Group at the Lawrence Berkeley National Laboratory and is an extensible, easily configured and programmed event-driven simulation engine, with support for several flavors of TCP (include SACK, Tahoe and Reno) and router scheduling algorithms. NS Version 2 extended Version 1's capabilities. Version 2's most significant modification was the incorporation of MIT's Object Tcl (OTcl) scripting language. OTcl is, in itself, an extension of John Ousterhout's Tool Command Language, better known as Tcl (pronounced *tickle*). OTcl allows for a finer grained object decomposition within the simulator. The details of this object decomposition will be described in detail.

B. DESCRIPTION AND INTERNALS

1. Description

NS is a stochastic, object oriented, discrete event simulator. It is an extensible, easily configured and programmed event-driven simulation engine specifically developed to support development and testing of network protocols and their interactions. The code for the simulator is entirely open-source.[1] The simulator is written in C++ code, but also incorporates OTcl classes and methods to be used as an interpreter. This allows interaction with the simulation through use of Tcl scripts and commands. Users create and manipulate simulator objects through the interpreter which instantiates objects and invokes methods in a C++ class hierarchy.

The objects instantiated in the OTcl class hierarchy are closely mirrored in the C++ hierarchy. It is this duality of objects that allows the user to manipulate and control the simulation through the Tcl scripts and commands. The use of Tcl also allows for the programming of arbitrary actions within a simulation. This is useful for simulating events like packet loss and packet delay during a simulation and is accomplished by the user by manipulating instance variables within the interpreter. [FALL99]

[1] The source code for all elements of the NS network simulator are available from the NS Website URL: http://www-mash.cs.berkeley.edu/ns/

2. Schedulers

The simulator is single threaded and only one event can be in execution at any given time. The scheduler selects the next event, in order, and executes it to completion. The scheduler then returns to execute the next event. No partial execution of events or preemption of processes is supported. If multiple events are scheduled to execute at any given time they are first ordered and executed one at a time. This ordering is dependent upon the scheduler in use. Event handling is scheduled through use of one of four different schedulers. The current schedulers available are a single linked list scheduler (default), a heap scheduler, a calendar queue scheduler and a real-time scheduler. The two schedulers of interest for this thesis are the single linked list scheduler and the real-time scheduler.

The single linked list scheduler operates as one would expect, all events are scheduled by appending them to a linked list and then are executed in order, First In First Out (FIFO). The real-time scheduler is used to introduce a simulated network into a real-world topology. This allows for emulated network nodes operating remotely from the simulation to interact with the simulator. The scheduler attempts to synchronize the execution of events with real-time. This scheduler is still under development and works for only relatively slow network traffic data rates. It will require before it can be incorporated into the SAAM simulation model.

3. Addressing

Although NS appears to support IP v6 addressing formats, in reality the mechanism for addressing of network entities is implemented using a unique scheme. Network addresses in NS, that is addresses of individual agents assigned to a node, are assigned by using unsigned 16-bit integers. The first eight bits represent the node *id_*. The second eight bits identify the specific agent at that node. When a packet reaches a node its header information is accessed by an address classifier. This classifier inspects the destination field of the packet header and determines if the packet is destined for that node by inspecting the first eight bits. If the first eight bits correspond to the node's *id_* it is then passed to a port classifier referred to as *dmux_*. The *dmux_* classifier then matches the second eight bits to the appropriate port corresponding to an agent at the node and passes the packet to that agent. If the packet is not destined for that node it is forwarded to the next node in the path according to the routing table at that node.

C. ARCHITECTURE

The NS architecture is complicated and extensive. Comprehension of the intricate linkage between the C++ class hierarchy and the OTcl hierarchy is essential to understand the methodology behind NS's inner workings. In an attempt to elucidate the architecture this thesis will describe several of the more important classes in both the C++ class hierarchy and the OTcl class hierarchy. Interactions between the hierarchies will be defined where useful and bindings between the OTcl instance variables and C++ instance variables will be discussed. In addition, methods for creating new classes in either

hierarchy will be presented along with procedures for creating class interactions. The easiest way to describe the architecture is to walk through an example simulation and discuss the NS classes and interactions as they are encountered[2].

When the simulator is invoked it creates a shell, much like tcsh or corn, that can be used much like any other shell. Interaction with the shell is done by way of Tcl commands or a pre-written Tcl script. Figure 3 is an example of such a script and will be the basis for a description of the NS architecture. Statements preceded by the "#" symbol are comments. The first line of the script is a declaration.

set ns [new Simulator]

In this case *ns* is the variable being declared. In OTcl brackets,"[]", are used as precedence indicators. The full meaning of this line is; Instantiate a new Simulator instance and assign it to the variable *ns*. The Simulator instance is an object of the Class Simulator class. The initialization procedure for this class performs several operations. It initializes the packet formats, which will be utilized throughout the simulation. The second operation it performs is that it creates a scheduler. The default scheduler is the linked list scheduler. The final initialization procedure creates a null agent, which is used as a packet sink used in various places.

[2] In order to follow this discussion a familiarity of OTcl syntax is required. An adequate tutorial of the OTcl language is provided in the download of the source code and is presented in Appendix D.

```
#Create a simulator object
set ns [new Simulator]

#Create three nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n0 $n1 1Mb 10ms SFQ

$ns cost $n1 $n2 1
$ns cost $n0 $n2 1
$ns cost $n0 $n1 2

#Create a CBR agent and attach it to node n0
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005

#Create a CBR agent and attach it to node n1
set cbr1 [new Agent/CBR]
$ns attach-agent $n1 $cbr1
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005

#Create a Null agent (a traffic sink) and attach it to node n2
set null1 [new Agent/Null]
$ns attach-agent $n2 $null1

#Create a Null agent (a traffic sink) and attach it to node n1
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

#Connect the traffic sources with the traffic sink
$ns connect $cbr0 $null0
$ns connect $cbr1 $null1

$ns rtproto Static

#Schedule events for the CBR agents
$ns at 0.5 "$cbr0 start"
$ns at 1.0 "$cbr1 start"
$ns at 4.0 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"

#Run the simulation
$ns run
```

Figure 3. Sample Simulation Script

The new simulator object provides numerous instance procedures, which fall into three categories: methods to create and manage the topology, methods to perform tracing functions and helper methods. The initialization procedure for this class performs several operations. It initializes the packet formats, which will be utilized throughout the simulation. Then it creates a scheduler. The default scheduler is the linked list scheduler.

Finally, the initialization procedure creates a null agent, which is used as a packet sink used in various places.

The next series of commands is the building of the topology of the network to be simulated.

set n0 [$ns node]

This command is also a declaration. The command inside of the brackets calls the Simulator Class instance procedure *node*. This procedure creates a new object of the Node Class. There are dual Node classes, one in the C++ class hierarchy and one in the OTcl class hierarchy. The instantiation of the OTcl object automatically triggers the creation of the C++ object. This is typical of most objects created through the OTcl interpreter. The Node Class is the basic building block of topologies within the simulator. An object of this class holds variables that reflect state information. This state information includes a reference to the simulator object, an array of neighbors, an array of agents attached to the node and a node identification.

In addition, the instantiation of a node creates two classifiers for that node. The first classifier is an address classifier. This classifier is used to inspect the destination field in the packet header to determine if the incoming packet is destined for this node. If the packet is destined for a different node then the classifier directs the incoming packet to the proper downstream node. The second classifier is a port classifier. If the packet's destination address is the same as the node's address then the address classifier passes the packet to the port classifier. The port classifier extracts the port address from the packet

and directs the packet to the proper agent at that node. The structure of a typical node is depicted in Figure 4.

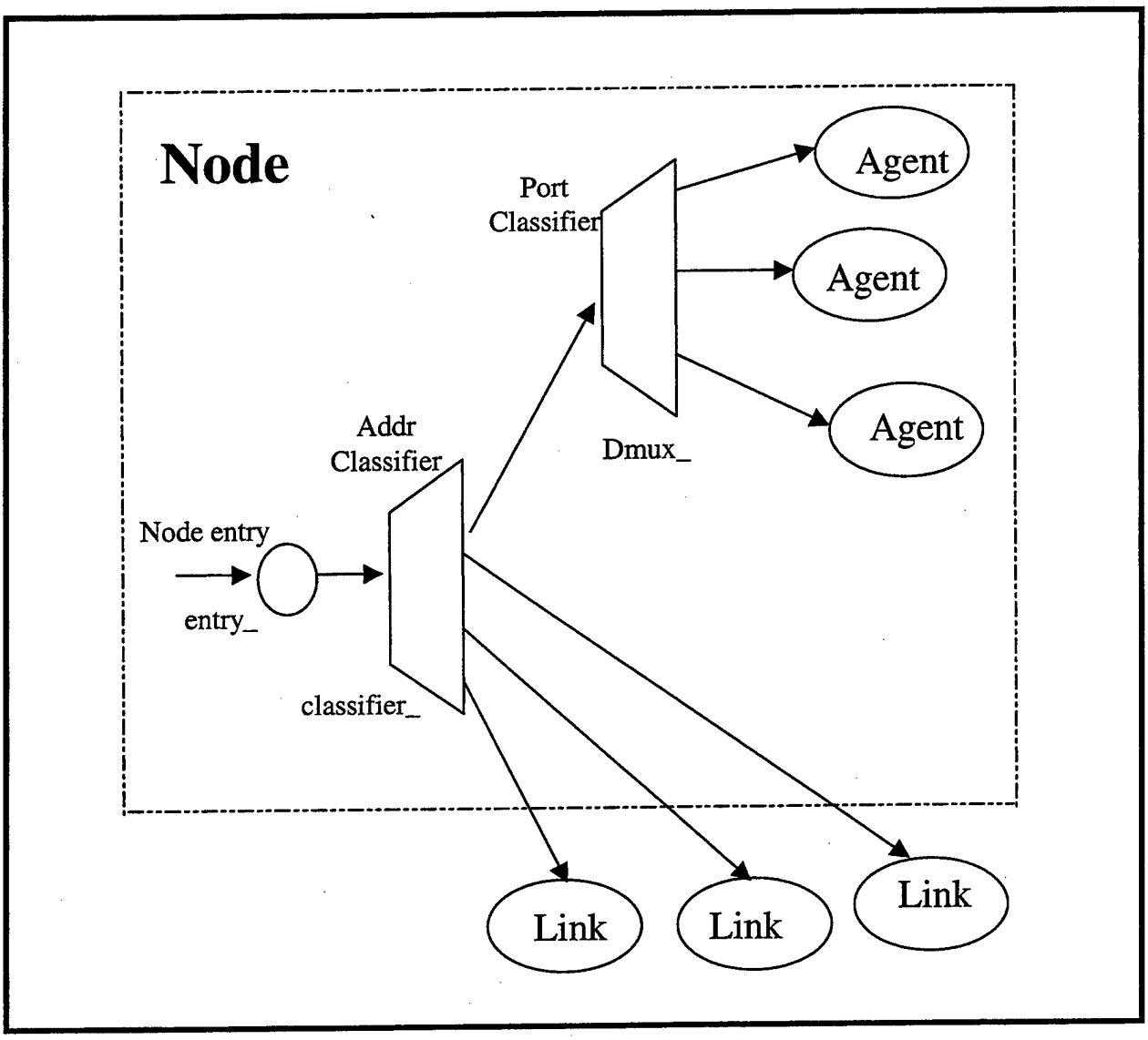


Figure 4. Structure of a Typical Node

The next step in building the topology is connecting the nodes with links.

$ns duplex-link $n0 $n2 1Mb 10ms DropTail

This line creates a duplex link between node n0 and n2. There is also an option to create a simplex link if that is part of the topology to be modeled. This line calls another of the

Class Simulator instance procedures, *duplex-link*. This instance procedure instantiates a link object of the Class Link class, which is implemented entirely in the OTcl class hierarchy. The link object maintains the bandwidth and the propagation delay of the link. In addition, it allows the modeler to designate what type of queue the link will utilize. In this case a drop-tail queue is selected. Other types of queues that can be used include Stochastic Fair Queues (SFQ), Weighted Fair Queues (WFQ) and Deficit Round Robin queues (DRR).

The next set of commands in the simulation script assigns costs to each of the links.

$ns cost $n1 $n2 1

It is important to note that these are one-way costs and different cost values can be assigned to a link depending on which way a packet is traveling on a link. Cost is a generic metric controllable by the modeler. The cost value is used by routing algorithms to determine proper path selection. Routing mechanisms utilized in the simulation will be discussed later.

The next four blocks of commands in the simulation script deal with the creation, assignment and configuration of agents. Agents are the most important factors for determining the behavior of simulations. They determine how simulation packet events are created, sent, received and handled. Agents often represent endpoints where network layer packets are constructed or consumed. Agents are also used in the implementation of protocols at various layers.

Agents are implemented from classes in both the C++ hierarchy and the OTcl hierarchy. The C++ class *Agent* maintains data members that keep internal state and are assigned to fields in simulated packet headers before the packets are sent. In the

definition of each class the type of packet header that class can create is also defined. The data members of the C++ class *Agent* can only be manipulated by a Tcl script if they have been linked to corresponding instance variables in an OTcl agent object. This linking is done using the *bind* command. The *bind* command links the C++ variable with the OTcl instance variable so that any change in one will be automatically reflected in the other. If a C++ variable is not bound to an OTcl variable then it can not be set, changed or manipulated by the simulation script.

In a simulation model an agent is created and then attached to a node. The agent is assigned a port address automatically based on what order it and other agents were attached to that node. Packets addressed to an agent will carry this port address appended to the node's address. Each portion of the address, the port number and the node address, is represented by one byte. The entire address is sent as an unsigned 16-bit integer, but the two halves are accessed separately by use of predefined offsets. All fields in packet headers are accessed in this manner. In the packet definition the offsets for each packet header field are declared. When the packet header is accessed the offset for the field being looked at is used to extract the values for that field.

From the simulation script the lines that create a CBR (Constant Bit Rate) agent and attach the agent to a node appear below.

```
set cbr0 [new Agent/CBR]
$ns attach-agent $n0 $cbr0
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
```

The first line instantiates a new *CBR* agent and assigns it to the variable cbr0. CBR stands for Constant Bit Rate and refers to a generic packet generator agent that emits a

stream of packets at a constant interval. The next line calls another of the Simulator Class instance procedures, *attach-agent*. This appends the newly created agent to node n0's array of assigned agents. The last two lines are assignment statements; *packetSize_* and *interval_* are instance variables of the newly created object and values are being assigned to those variables.

In order for an agent to transmit packets it must be connected to another agent. The address of the packet receiver is stored in the packet generator's *dest_* instance variable. The next block of commands creates a generic packet receiver called a sink or null agent.

```
set null1 [new Agent/Null]
$ns attach-agent $n2 $null1
```

These two lines create a new null agent, assign it to the variable null1 and then attach the agent to node n2.

The next step is to connect a packet sender to a packet receiver. This is done in the next series of commands in the simulation script.

```
$ns connect $cbr0 $null0
```

This line sets the destination instance variable in the CBR agent to handle for the null agent and visa versa. It is important to note that packets are sent between agents by the sender calling a procedure that belongs to the receiving agent. The simulated routing of that packet is done separately by specialized routing agents attached to each node in the simulation.

At this point the topology for the simulated network is complete. A graphical representation of this topology is presented in Figure 3. Circles represent simulated

nodes with their node identifiers centered. Agents attached to each node are in boxes and connected with dashed lines. The duplex links between nodes are depicted using solid black lines and the associated costs are annotated next to each link.

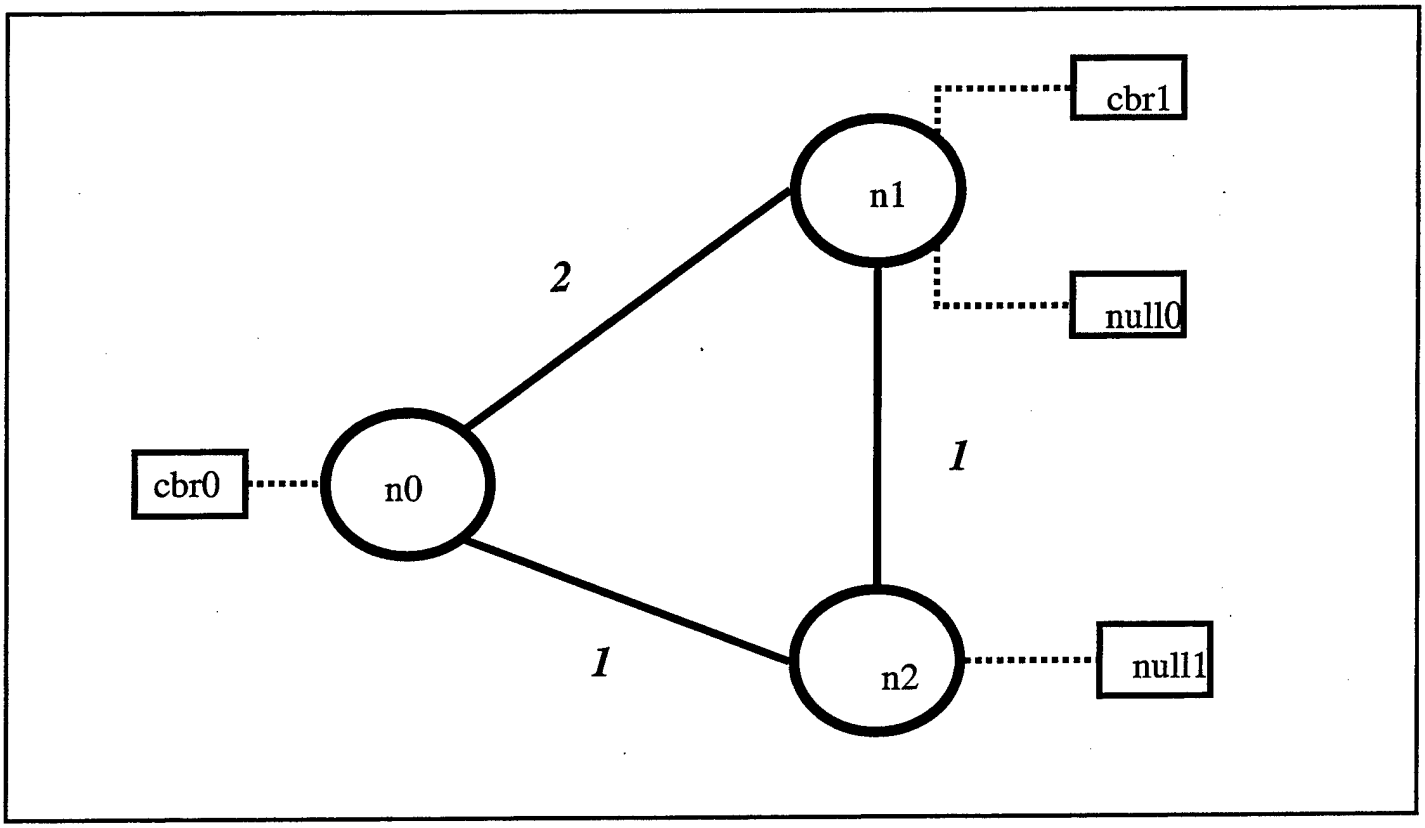


Figure 5. Sample Simulation Topology.

Now that the topology has been set, a routing capability needs to be added. This is done in the next line.

$ns rtproto Static

This line represents another call to a Simulator instance procedure. In this case the instance procedure is *rtproto*. The *rtproto* instance procedure provides the modeler a mechanism for specifying the desired routing strategy to be implemented by selecting a

protocol, which is used as the argument for the instance procedure. The current version of NS provides three different routing protocols; static routing, session routing and distance vector routing.

Static is the default routing protocol. If no *rtproto* command is given in the simulation script, then static routing is implemented for all nodes in the simulated network. When static routing is used all routes are calculated once, prior to the beginning of the simulation. If the topology changes then nodes may become unreachable from some sources. It is an implementation of Dijkstra's all-pairs shortest path first routing algorithm. The routes are computed using an adjacency matrix and link costs of all the links defined in the simulated topology.

The session routing protocol is very similar to the static routing protocol. Both use identical methods to compute routes. Like static routing, session routing computes routes prior to the beginning of the simulation. Session routing differs from static in that if a change in topology occurs during a simulation, session routing will re-compute routes based on the new topology. Routes are re-computed utilizing the same algorithm as the original computation.

Both static and session routing are centralized routing schemes. That is, they both utilize one global routing table to make routing decisions and forward packets. The other option is to incorporate a decentralized routing scheme. In NS this is considered a dynamic routing strategy. In dynamic routing the routing tables are maintained at each node. In addition, dynamic routing allows for more than one routing protocol to be implemented at each node. The only dynamic routing protocol included with the standard

NS distribution is the Distance Vector routing protocol. The Distance Vector routing protocol utilizes the Distributed Bellman-Ford algorithm.

Once the topology is defined and a routing scheme is established the simulation is ready to start scheduling events. This scheduling is accomplished using the Simulator instance procedure at as illustrated below.

$ns at 0.5 "$cbr0 start"
$ns at 1.0 "$cbr1 start"
$ns at 4.0 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"

The *at* procedure takes a simulated time and a string representing a command as arguments. The end result is the simulator evaluates the string as a command and schedules its execution at the prescribed time. The first line from above evaluates to: start agent cbr0 at simulation time 0.5 seconds. This initiates the CBR agent, cbr0, sending of packets to its prescribed destination at the prescribed interval until the *stop* command is scheduled at simulation time 4.5 seconds.

The final command issued from the simulation script is *run*. The *run* command actually initiates the simulation. It begins executing all commands in accordance with the topology and parameters given. Tracing mechanisms are also available, but are beyond the scope of this discussion. Once all commands are executed and there are no events remaining in the scheduler the simulation is complete and the program exits.

D. WHY NS?

The NS simulation tool was selected for a number of reasons. The single most important factor was that NS is one of only a few open source network simulation

packages available. Finding an open source simulation package to model SAAM was essential in the development of new protocols and implementing new behaviors in the simulation. In order to model centralized routing decision making and decentralized packet forwarding, as in the SAAM architecture, new routing mechanisms and node behavior had to be developed. Without the ability to modify an existing package, this would not be feasible.

A second reason for the selection of NS was its stability. NS has a substantial history and has been under constant development for nearly ten years. It has been widely accepted throughout the network research community. In addition, the core functionality and architecture has been extensively validated and verified and subsequent additions are not allowed until appropriate validation and verification has been conducted. This level of assurance is paramount when attempting to develop new architectures. The new models developed in NS still require validation and verification, but having these tasks accomplished on the core of the simulation package already significantly simplifies this process.

The final criteria for selecting NS was its robustness. Many existing protocols and behavior have already been modeled using NS and have been incorporated into the standard release. For SAAM to receive wide acceptance it will have to be shown that it can be seamlessly integrated with existing protocols in use. These protocols include existing network and transport layer protocols, many of which have already been incorporated in NS.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. THE MODEL

A. GOALS

The largest goal in the designing of this model was to be able to simulate the dynamic behavior of control message traffic within a SAAM architecture. In order to accomplish this the model had to first emulate the architecture, which meant being able to produce expected behaviors through use of a simulation tool. The first control channel behavior to be modeled was simply being able to pass control messages to and from simulated nodes across a control channel. The first step was creating the packet headers of the applicable control messages. Next specialized nodes had to be created. These nodes were to interpret and use the control messages. Specialized nodes are constructed by attaching agents to the nodes. The agents actually dictate the behavior of the node.

The agents to be created represented the SAAM server, router, and a specialized application, which utilized a flow request to register its network traffic. The server had to maintain a PIB from which to calculate routing information. It also had to be able to receive and act on flow requests and LSA's. In addition the server had to be able to send Flow Routing Table (FRT) updates to routers on a specific flow path. The router had to be able to generate and send LSA's to the server and receive FRT updates from the server in order to make appropriate changes to its FRT to accommodate a newly admitted flow.

Once the agents controlling the specialized nodes were created, flow based routing needed to be implemented in order to emulate the mechanism that SAAM uses to route packets. This required the creation of a new *rtproto* agent to handle the specialized

routing mechanism. This operation in particular required an in depth study and modification of the existing NS source code and an intricate understanding of the NS architecture.

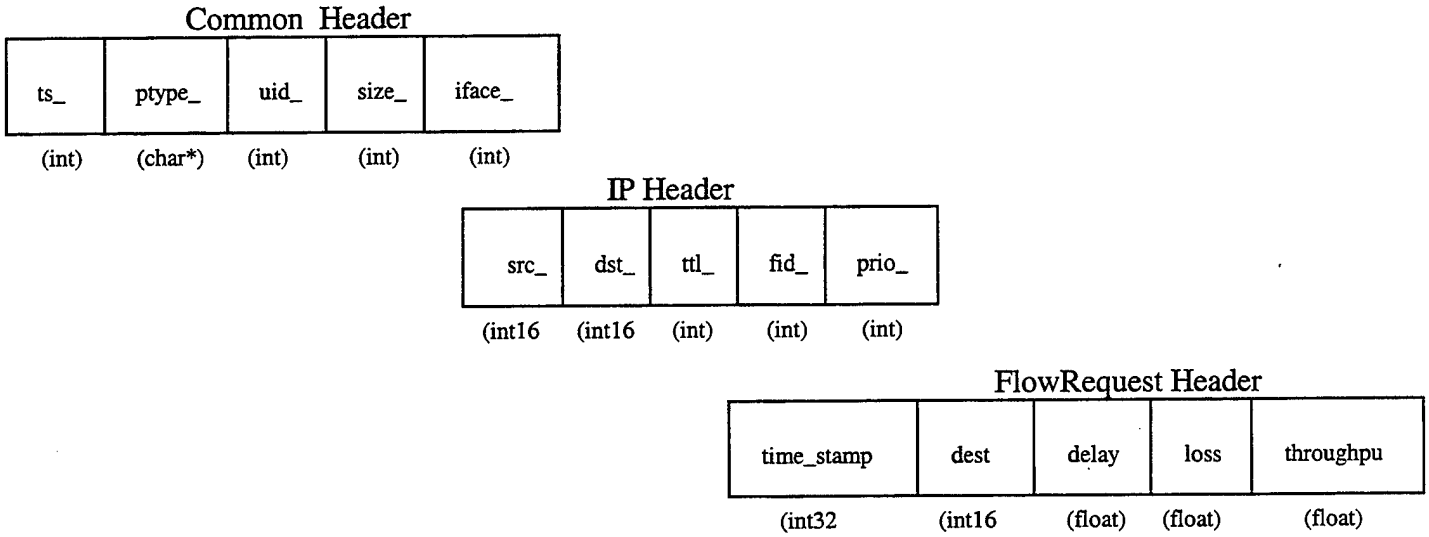
The remainder of this chapter describes in detail the new classes and objects that had to be created in the making of this model. In addition, explanation of the modification of existing NS source code will be provided. The chapter will be broken down by the agents that were created. Specifically, the chapter will address the creation of the *SAAMServer/Agent,* the *SAAMRouter/Agent,* and the *SaamApp/Agent.* References to the source code will be made where applicable. The new and modified NS source code can be found in Appendices C – E.

B. PACKET FORMATS

There are five control messages of interest in this model. They are flow requests, flow responses, flow routing table adds, flow routing table deletes, and link state advertisements. Each control message is responsible for carrying information important for the proper configuration of the SAAM routing mechanism. They can be grouped by the type of specialized node where they originate. The flow request is generated by the application and is sent to the server. Messages which originate from the server, flow responses, flow routing table adds and flow routing table deletes are grouped into a single packet format called SIF's (Server Initiated Functions). LSA's originate at the router.

The flow request initiates the model's configuration process. Sent by the application to the server the flow request contains the information pertaining to the QoS parameters that the application requires for its network traffic. These parameters include

packet delay, packet loss and throughput. In addition the header contains a time stamp and the flow destination. Because all traffic in the simulation is carried in IP packet format, the flow request packet is carried as part of the IP data payload. The flow request packet carries all of its information in the form of the header and has no real payload. The packet header structure is defined in the file associated with the *SaamApp/Agent*, "saamapp.h". The structure of the flow request packet is displayed below.



The Common Header and the IP Header are generic to NS packets. The Common Header contains a time stamp, a packet type, a unique Id, packet size, and an interface identifier. The packet type field is to alert the receiving classifier what type packet is arriving so that the receiver can use the appropriate offsets to access the header information. The IP header fields are an abbreviated form of the standard, note that it includes a flow Id and a priority field which are IP v6 specific. It should be noted, also, that the destination field in the IP Header refers to the destination of the packet, where as the destination field of the Flow Request Header refers to the ultimate destination of the flow for which the application is requesting.

Server Initiated Function (SIF) packets are defined in the "saamserver.h" file. One packet format is sufficient to accomplish all requirements of messages originating at the server. The packet format is displayed below. Note that the Common Header and IP Header have been omitted.

SIF Header

time_stamp	sif_type	fid	next_ho	sl	flow_request_id
(int32	(int)	(int)	(int16	(int)	(int)

The *sif_type* field identifies which type of SIF the packet contains. A value of 1 in this field identifies the SIF as a Flow Routing Table Add, which is used to add an entry to a router's flow routing table. A value of 2 identifies the SIF as a Flow Routing Table Delete. A Value of 3 identifies it as a Flow Response.

The *fid* field is the Flow Id. It is generated by the server once a Flow Request is received. It is used by the requesting application to fill in the Flow Id field in the IP packet header for all packets in the flow once the flow has been established. The *fid* field is used by the router to index entries in the Flow Routing table. The *next_hop* field identifies the next hop a router should use to forward packets with the corresponding *fid*. The *sl* field identifies the service level pipe the server has identified that supports the requesting application's QoS requirements. The *flow_request_id* is used by the application to match the flow response with the appropriate flow request.

LSA's are defined in the file "saamrouter.h". The packet format is displayed below.

LSA Header

time_stamp	origin	endpoin	delay	loss	throughpu
(int32	(int16	(int16	(float)	(float)	(float)

The *origin* and *endpoint* fields are NS addresses and are used to identify the link that is being described by the LSA. The *delay*, *loss* and *throughput* fields reflect the QoS parameters that are available on that link and are of the float data type. These QoS fields are used by the server to identify available network resources when assigning flows to a path. The server uses the information in the LSA to update its PIB, which is used to keep track of the state of the network as a whole.

Any new packet header class that is created in NS has to be registered in the file "packet.h". This is where the simulator keeps a record of all packet headers in the simulation. Upon initiation of the simulator, one of each type of packet is created and kept for reference by the packet manager. It is by this method that the simulator can de-reference the offsets for each field in a header.

C. APPLICATION

The SAAM routing mechanism is initiated when an application sends a flow request. To represent this action a new application agent had to be created within the NS simulator. *The SaamApp/Agent* is defined in the files "saamapp.h" and "saamapp.cc". The header file includes a structure for the flow request packet header. The *SaamApp/Agent* is inherited from the C++ class *Agent*. A new *SaamApp/Agent* is instantiated through the OTcl interpreter when the user declares one.

In this version of the model the *SaamApp/Agent* is only able to send Flow Requests and receive Flow Responses. In future versions of the model the application will utilize the information in the Flow Response to append it to its application flows. The *SaamApp/Agent* is able to send Flow requests by utilizing the *command* method as inherited from the *Agent* class. This allows the user to input a character string as a command to an object through the OTcl interpreter. The interpreter forwards the string to the C++ object and if the string is recognized as one of the developer defined command options it executes the command. In the case of the *SaamApp/Agent* the model has created a *launch* command under the *command* method. When *launch* is called the *SaamApp/Agent* allocates a new Flow Request packet. It then fills in the applicable header information. Currently the *SaamApp/Agent* fills in the *time_stamp* field and the *dst_* field. This is enough information to begin the SAAM configuration process.

The *SaamApp/Agent*'s *recv* method is used to act on incoming packets addressed to the application. When a packet arrives at the application, the *recv* method is automatically called. In the case of the *SaamApp/Agent* the *recv* method accesses the IP and SIF headers. Currently the application only acknowledges receipt of the SIF packet. In future versions the application will use the information appropriately.

D. SERVER

The server is the workhorse of the SAAM architecture. In the model it is represented by the *SaamServer/Agent*. The *SaamServer/Agent* is defined in the files "saamserver.h" and "saamserver.cc". This agent also inherits from the C++ class *Agent*. When an application sends a flow request to the server the server must be able to process

the request. The server is responsible for maintaining the Path Information Base (PIB). The *PIB* is an array of *PIB_entries*. A *PIB_entry* is a structure which represents a complete path from a source to a destination to include the QoS parameters available across the path. The definition of the *PIB_entry* is shown below.

```
struct PIB_entry{
nsaddr_t        src;
nsaddr_t        dest;
nsaddr_t        nodes_in_path_array[4];
int             delay;
int             loss;
int             throughput;
};
```

The *PIB* contains a *PIB_entry* for each possible path from all sources to every destination in the simulated topology. The *src* and *dest* fields are of the type *nsaddr_t* which is declared as a 16 bit unsigned integer. Currently the model contains a static *PIB*, which is hard coded to reflect a single sample topology. Future versions of the model will require methods to dynamically build and modify the *PIB* based on changing topology information as gathered by the server.

Normally the *PIB* is used as the basis for making all routing decisions. The topology information reflected by the *PIB* is gathered by the server by either active agents or by the receipt of LSA's. In order to collect this information the server must be able to receive and process LSA's. The model does this through its *recv* method. The server's *recv* method must be able to differentiate between the different types of messages that the server may receive. In order to do that the *recv* method accesses the common header of the incoming packet. The *ptype_* field of the common header tells the server what type of packet it has just received and can then process it accordingly.

Once an incoming packet is recognized as an LSA, the server would access the header and use the information to update its *PIB*. Currently the *SaamServer/Agent* is only able to acknowledge the receipt of an LSA. Future versions will use the source and destination fields of the LSA to identify all paths, which include that link. It will determine if the QoS parameters of each of the affected paths need to be adjusted based on the LSA information, and then make appropriate changes to those *PIB_entries*. It is by this method that the model is able to demonstrate the adaptability of the SAAM architecture.

If the packet received is determined to be a flow request, then the server must be able to appropriately process that request. The server would first access the flow request header to determine the source and destination of the requested flow. Using this information the server would index its *PIB* to find a *PIB_entry* with the same source and destinations and whose QoS fields were greater than or equal to the requested flow parameters. The first flow, which met the parameters, would be selected and the flow assigned an Id. The server would then access the *nodes_in_path_array* field of the selected *PIB_entry*.

The *nodes_in_path_array* contains the Id's of all of the routers which are associated with the path identified by the *PIB_entry*. Using these Id's the server would generate and send Flow Routing Table Updates to each of the affected routers. Currently the model is not able to include the appropriate routing information in the Routing Table Updates. The model is, however, able to launch a Server Initiated Function (SIF) which is designated as a Routing Table Update to every router in the simulated topology in response to the reception of a Flow Request.

Once all affected routers have been updated the server sends a Flow Response to the requesting application. The Flow Response is a SIF and would normally include the *fid* and the *flow_request_id*. Currently the model is only able to launch a generic Flow Response message. Future versions of the model will include this information to inform the requesting application that its request has been granted, if available network resources exist.

In order for the server to be able to address and send messages to all routers in the topology it must first know of the router and second maintain a record of the router's address. The *SaamServer/Agent* maintains a *rtr_array*. The *rtr_array* is an array of router addresses. Currently the *rtr_array* is limited to 40 routers, but is easily expandable to accommodate larger topologies. Each time a router sends an LSA to the server the source address of the IP packet is accessed and compared to each element in the *rtr_array* until a match is found. If no match is found the address is appended to the *rtr_array*. In this way the server is able to maintain a record of all routers which have sent it an LSA. As part of the initialization of the simulation each router is required to register with the server by artificially connecting its agent with the *SaamServer/Agent* and launching an introductory LSA. This registration is accomplished through the simulation script.

In a method similar to router registration, each application also registers with the *SaamServer/Agent*. The applications are not required to register as a part of the initialization of the simulation. Each application is registered with the server upon the receipt of its Flow Request. The applications are recorded in the *app_array*. Currently the *app_array* is limited to 2 application addresses, but can be easily be expanded in the simulation for future versions.

E. ROUTER

In the model the router is represented by the *SaamRouter/Agent*. The router is responsible for implementing all of the routing decisions made by the server. It should also be able to monitor the state of all of its associated links. The actual forwarding of application traffic is accomplished by a separate agent, which must also be attached to any node designated as a router. The agent, which accomplishes the forwarding of packets, inherits from the *rtproto/Agent* and is called *SaamRtproto/Agent*. This specialized agent is defined in the files "rtProtoSAAM.h" and "rtProtoSAAM.cc" and will be discussed later. The *SaamRouter/Agent* is defined in the files "saamrouter.h" and "saamrouter.cc".

The *SaamRouter/Agent* is responsible for the sending and receiving of control messages. It must be able to receive the two types of SIF messages "Flow Routing Table Add" and "Flow Routing Table Delete". The *SaamRouter/Agent* is responsible for extracting the information from the server sent SIF's and passing that information directly to the *SaamRtproto/Agent*. The router is also responsible for generating LSA's and sending them to the server.

Currently the router is able to receive SIF's, extract the contained information and acknowledge receipt. The router is also able to generate LSA messages and update the *time_stamp* field. Once the LSA's are generated the router is able to launch them to the server. The model is currently unable to monitor the associated links in order to maintain

a picture of the network-state, but this is not required to model the message passing characteristics.

F. STRENGTHS AND WEAKNESSES

1. Strengths

Although the model is incomplete, it represents some invaluable progress in the modeling of the SAAM control channel dynamics. The model defines the structure of all control messages. These definitions include the data types and fields required to relay all pertinent information from object to object. The mechanism for the sending and receiving of these control messages is in place. This accomplishment in itself represents the establishment of the control channel and forms the basis for all future work in the model.

The model was built in an object-oriented design under the NS framework. The modularity of the model allows for ease of implementation and future design efforts, at least under the limitations presented by the NS simulation tool. The modeler with a reasonable understanding of NS can easily comprehend the design structure of the model and be able to extend its functionality with a minimum study period. The code is reasonably well commented and documented so that the logic implemented is easily followed and understood.

Where possible, duplication of packet fields has been avoided. Duplication does exist, but only to the extent that the duplication was required to carry information that

would have been lost as a packet travels up and down the protocol stack. If certain data would become consumed by the lower layer of the protocol stack and was of importance to the next layer, then duplication of this information was unavoidable. This is consistent with current practices, and where practiced represents more efficient use of resources than passing additional messages between protocol layers.

The greatest strength in this model is groundwork that has been constructed for future iterations and research. The creation of the basic objects and the establishment of the required C++ and OTcl linkages represent a monumental step forward in the development of a useable architectural model that will enhance the study and research of the SAAM project. A majority of the effort reflected in the development of this model was gaining an understanding of the inner workings and architecture of the NS simulation tool. Converting relatively simple conceptual designs to implementable model entities and behaviors was no small task and represented the most time consuming portion of the work effort in developing this model.

2. Weaknesses

The model in its current configuration is unmistakably incomplete. To completely model control channel dynamics a flow based routing mechanism is required. Unfortunately the effort to implement this mechanism has been heretofore unsuccessful. An even greater understanding of the NS simulation tool is required and the resident expertise to achieve that goal was not locally present or available in time to incorporate it in this version of the model. Future development in this area will be predicated on a more focused and in depth study of this particular facet of NS.

The current version of the model possesses no mechanism for discovery of nodes in the network. That is, there is no automated way for the server to receive information pertaining to the existence of routers or host nodes in the simulation. This lack of discovery results in the need for an awkward initialization process through the simulation script. This initialization requires that each router be artificially connected to the server agent in order to send an introductory LSA so that the router can be registered with the server and be added to the server's *rtr_array*. A simple version of a hello protocol may be useful for future versions to rectify this shortfall.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

A. LESSONS LEARNED

The model as presented in its current form does not reach the goals as set out at the beginning of this study, but many lessons have been gleaned in the development of the model thus far. These lessons learned fall into two broad areas. The model development area mostly consists of the intricacies involved in building simulation models using the NS simulation tool. The other broad area concerns the SAAM architecture and its implementation.

1. NS Development

In order to conduct development in NS the developer must know 3 different languages. A knowledge of C++ and Tcl is required along with an understanding of NS itself. NS is a difficult and convoluted software package, which takes a great deal of study to comprehend. Although there are volumes of information available to support the NS developer, none of them seem to offer a clear and understandable description of the architecture. The most informative (and confusing) of the material is contained in the NS Notes and Documentation. [FALL99] This document contains a detailed, but disjointed, technical description and users manual.

Even with all of the so-called help available for NS there is no substitution for getting your hands dirty in the code. It took a great deal of reading, searching and experimentation with the NS code before any serious development could begin. Though

the NS learning curve is very steep, it is still a very promising research tool and should be considered for any further work in the SAAM modeling effort.

Selecting a development platform was also an initial stumbling block in the model building effort. Originally it was planned to accomplish the modeling effort on a standard Personal Computer architecture using the Microsoft Windows NT operating system. According to the NS web page (URL: http://www-mash.cs.berkeley.edu/ns/ns-build.html) the simulation tool can be compiled and run from this architecture. However, after many weeks and numerous attempts, successful compilation was never accomplished. Different choices of operating systems were tried. The ultimate choice of operating systems was the Linux operating system. Several versions of the Linux operating system were used to compile and run NS and no significant problems were discovered on any of them. Documentation for downloading, unpacking and compiling NS is available in Appendix A.

A significant roadblock to designing an object-oriented model in NS is that the tool itself is not completely object oriented. It is written in a combination of object-oriented design and functional programming. The shear volume of source code associated with the simulation tool is daunting, but the convoluted aggregation of functional code, object oriented design, C++ code and Tcl code presents a conflagration which is not easily deciphered. With this model as a basis, however, extension of this concept should prove more fruitful in future versions and a useful more robust model can be developed with a minimum of effort.

2. SAAM Development

Throughout the course of this study great strides have been made in the further refining of the SAAM concept as a whole. Definitions of packet structures, required objects, use cases and object interactions received great consideration and study. This thesis was developed in parallel and in conjunction with another work pertaining to the SAAM architecture. Marine Corps Captains Dean Vrable and John Yarger have accomplished an in depth study of the SAAM architecture as a whole and developed an application layer emulation of both the SAAM server and the SAAM router. Their work incorporates many of the lessons learned and would be a beneficial starting point in the study of the SAAM architecture. [VRAB99]

The conceptualization and development of the PIB and the flow routing tables were two of the more noteworthy accomplishments of the joint effort provided between the two studies. Though the full realization and conceptualization of these concepts are not completely reflected in this thesis, their contribution cannot be ignored and represent a monumental step forward in the final realization of the goals set out for the SAAM project as a whole.

The control channel dynamics that this thesis proposed to model are also important factors, which will ultimately determine the success of the SAAM project. Determining the timing of when LSA's should be initiated by the router is a fundamental question. Use of triggers and hold-down timer will present a realistic and implementable solution to this question. However, much study and experimentation is still required to determine the change thresholds and hold-down timer values that will provide the most effective and efficient control configuration.

B. FUTURE WORK

Though this model represents a great step forward in the SAAM research, there is still much left to be done to create a viable model worthy of running detailed simulations for architecture development and testing. Many of the stated goals for this thesis remain unfulfilled and require future work. The remainder of this sub-section identifies many of the areas requiring future development. Though the following list is not all-inclusive, the author feels that these areas will provide the most significant impact on extending and improving upon this work.

1. Flow Based Routing

The greatest hurdle left in the development of this model is the implementation of a flow based routing scheme. Until a viable flow based routing scheme is implemented further exploration of control message causality cannot be explored. In order to validate the effect of control channel messages on the configuration of the system, the system must be able to route network traffic packets. Without this ability there is no way to test the validity of the concept or conduct any worthwhile experimentation with the components that have been developed in the current model.

To implement flow based routing in this model a further study into the NS routing mechanism is required. In specific, the study needs to concentrate on the dynamic routing algorithms already implemented in the NS architecture. Particular attention needs to be paid to the *rtProto* agents, the *route-logic* objects and their implementations of routing

tables and lookup procedures. Examples of these agents can be found in the files "rtProtoDV.h" and "rtProtoDV.cc". In addition the Tcl files which define the instance procedures to manipulate these objects through the OTcl interpreter are "route-proto.tcl" and "ns-route.tcl". Focus on these areas and development of flow based routing schemes, may in themselves present a complete topic for future study.

2. Dynamic PIB Configuration

The current model implements a test topology based upon a static PIB "hard-coded" into the *SAAMServer/Agent*. To fully realize the proposed SAAM architecture the PIB must be able to incorporate topology changes as presented to the server based on LSA's received from individual routers. These changes should not only reflect the connectivity diagram maintained by the server, but should also include the QoS parameters associated with each link.

The PIB is the basis for all routing decisions made by the server and should reflect the most up to date information available to the server. Implementing dynamic updating of the PIB should not present the developer with a major challenge. It simply entails extracting the information from a received LSA and making the appropriate change to the corresponding *PIB_entry* in the PIB. The challenge will lie in the proper selection of an appropriate data structure for the PIB. Latency of updates and lookups in the PIB represent a major bottleneck in the performing of the server's mission. Reducing this latency will greatly enhance the overall speed of the model in subsequent simulations.

3. Service Level Pipes

The implementation of service level pipes into the model is probably the most challenging endeavor left for the successful modeling of the SAAM architecture. The concept of service level pipes represents the single most important factor for the implementation of QoS routing in SAAM. To implement service level pipes into the model will require determining how to partition individual links. The next step is to develop a method for actually assigning network flows to an individual service level pipe. One concept developed to solve this problem is in the use of individual outbound queues for each service level pipe.

To ensure that each link maintains its QoS guarantees schedulers will have to be developed to manage the delay and loss rate at each queue. To maximize available resources, service level pipes that are not being fully utilized should be made available to traffic from other service levels as long as the additional packet flow doesn't degrade the QoS of the traffic assigned to that pipe. Queue management and the development of the required schedulers represent a large developmental effort and are the topic of another individual study.

C. SUMMARY

Though not all of the goals of this thesis have been met, it undoubtedly represents a significant step towards the successful modeling of the SAAM architecture. Much has been learned in the effort that will undoubtedly progress the project as a whole. The SAAM project represents a major conceptual breakthrough in the effort to realize the

Next Generation Internet. Much is left to be done before implementation of SAAM can be accomplished, but the research already completed lays the groundwork for future accomplishments. Regardless of the ultimate success or failure of the project, the research being conducted in this effort is applicable and beneficial to the research community. Many of the concepts, which SAAM has proposed, represent "out-of-the box" thinking and will without a doubt result in a paradigm shift in the network research field. For the author this has been a worthwhile experience and it is his hope that he may continue in this ongoing effort.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. [NS DOWNLOADING AND INSTALLATION PROCEDURES]

This brief tutorial is a step-by-step instruction list for downloading and installing the NS network simulator on a standard IBM compatible personal computer. These particular instructions reflect steps required to install the simulator on a computer utilizing the LINUX operating system. The computer utilized for purposes of this tutorial was a dual Pentium 300 architecture with 256 Mb of RAM. The system was dual booted with Microsoft Windows NT 4.0 and the Slackware variant of LINUX. Differences in configuration of computers may impact on installation procedures.

The entire source code for NS including all required and optional packages are available at URL: http://www-mash.cs.berkeley.edu/ns/. Generic installation procedures and specific trouble shooting guides based on different architectures are also available at this location. The minimum required download consists of the following files:

ns-src_21b5_tar.tar
otcl-1_0a4_tar.tar
tclcl-src-1_0b8_tar.tar
tcl8_0_4_tar.tar
tk8_0_4_tar.tar

STEP 1: Download required source code.

There are two options for downloading the source code. All packages, required and optional, can be downloaded using the all-in-one download option. The other option is to download only those packages required one at a time. Whichever option is selected, source code should be downloaded into one main directory.

STEP 2: Unpack the source code.

The source code when downloaded is compressed in a tar format. In order to be utilized it must be first unpacked. Unpacking is done at the command prompt utilizing the tar command with the –xzvf options. Below is an example of a typical unpacking command.

tar –xzvf <file name>

The result of this command is to unpack all of the associated files into their own subdirectory. This needs to be done for every file downloaded from the website.

STEP 3: Make the Executable for each package and Install

This is where it may get a little tricky. The instructions on the website are terse and sometimes incomplete. It is important to install all of the packages in order. The first package to install is the Tcl package. The following steps are required to compile and install the Tcl package. Begin in the Tcl directory.

1. Change to the *unix* sub-directory type *cd unix*
2. From the *unix* sub-directory type *./configure*
3. Now compile the package type *make*
4. Install the package type *make install*

If any problems arise during this phase there are *README* files in both the Tcl directory and the unix sub-directory, which may help in resolving conflicts.

The next package to install is the Tk package. The following steps are required to compile and install the Tk package. Begin in the Tk directory.

1. Change to the *unix* sub-directory type *cd unix*
2. From the *unix* sub-directory type *./configure*

3. Now compile the package type *make*
4. Install the package type *make install*

There are also *README* files in both the Tk directory and the unix sub-directory to assist in resolving compilation and install errors.

The tclcl package is the next to be installed. Start in the *tclcl* directory.

1. Directly from the *tclcl* directory type *./configure*
2. Compile the package type *make*
3. Install the package *make install*

Next the OTcl package must be compiled and installed. Start in the *otcl* directory.

1. Directly from the *otcl* directory type ./configure
2. Compile the package type *make*
3. Install the package type *make install*

Finally you are ready to compile the NS simulation tool package. Start in the *ns* directory.

1. Directly from the *ns* directory type *./configure*
2. To compile type *make*
3. A test package is available to verify that all packages and dependencies are correctly installed and compiled. type */validate*. Be prepared as the validation programs may take up to 20 minutes to complete.

When the validation programs complete without error you have successfully compiled and installed all requirements needed to use the NS simulation tool. It is important to remember that if you alter the source code or add additional files and need to recompile, it is first required to delete all of the previous NS object files. To do this type *make clean.*

Then prior to re-compiling be sure to recreate any needed dependencies by typing *make depend.*

Extensive additional help to resolve installation and compiling errors is available at the NS homepage URL -- http://www-mash.cs.berkeley.edu/ns/ns-build.html

APPENDIX B. [DEVLOPMENT STRUCTURE]

Below is the directory structure for this development;

NS_HOME = /nsdev/ns-2.1b5
TCLCL_HOME = /nsdev/tclcl-1.0b8
OTCL_HOME = /nsdev/otcl-1.0a4
TCL_HOME = /nsdev/tcl8.0
TK_HOME = /nsdev/tk8.0

In this directory structure */nsdev* is the original directory where the original compressed source files were downloaded. Subdirectories of */nsdev* were created when these original files were unpacked as described in appendix A. This directory structure will be used when describing file location throughout the remainder of this appendix.

1. SAAM EXECUTION SEQUENCE STARTING WITH A FLOW REQUEST

This section will describe the SAAM execution sequence that starts with an application sending a Flow Request to the server. As can be seen in Figure 6 the SAAM execution sequence begins once an application requiring QoS sends a Flow Request to the server. The code required to accomplish this can be found in "NSHOME/SaamApp.cc". The actions which take place at the server can be found in "NSHOME/SaamServer.cc". The pseudo code shown in the boxes represent the code that actually executes these actions. The sending of a Flow Request is initiated through the Tcl simulation script. The syntax for this action looks like this;

$ns at 0.2 "$app1 launch"

The flow request would contain the bounds on packet loss, delay and throughput that the application was requesting along with the source and destination of the requested flow. It is sent using the *command* method in the *saamapp/agent*.

Upon the Flow Request's arrival at the server the *recv* method is called and the *SaamServer/Agent* will begin processing the request. First the server must access the Flow Request packet header and be able to extract the necessary fields. Using the *src* and *dest* fields the server will index the *PIB* to find a *PIB_entry* which contains the link between *src* and *dest* and has values in the *loss*, *delay* and *throughput* fields which are greater than or equal to those requested. The server then generates and sends Flow

Routing Table Add (one of the *sif_types*) messages for each router contained in the *nodes_in_path_array* of the selected *PIB_entry*.

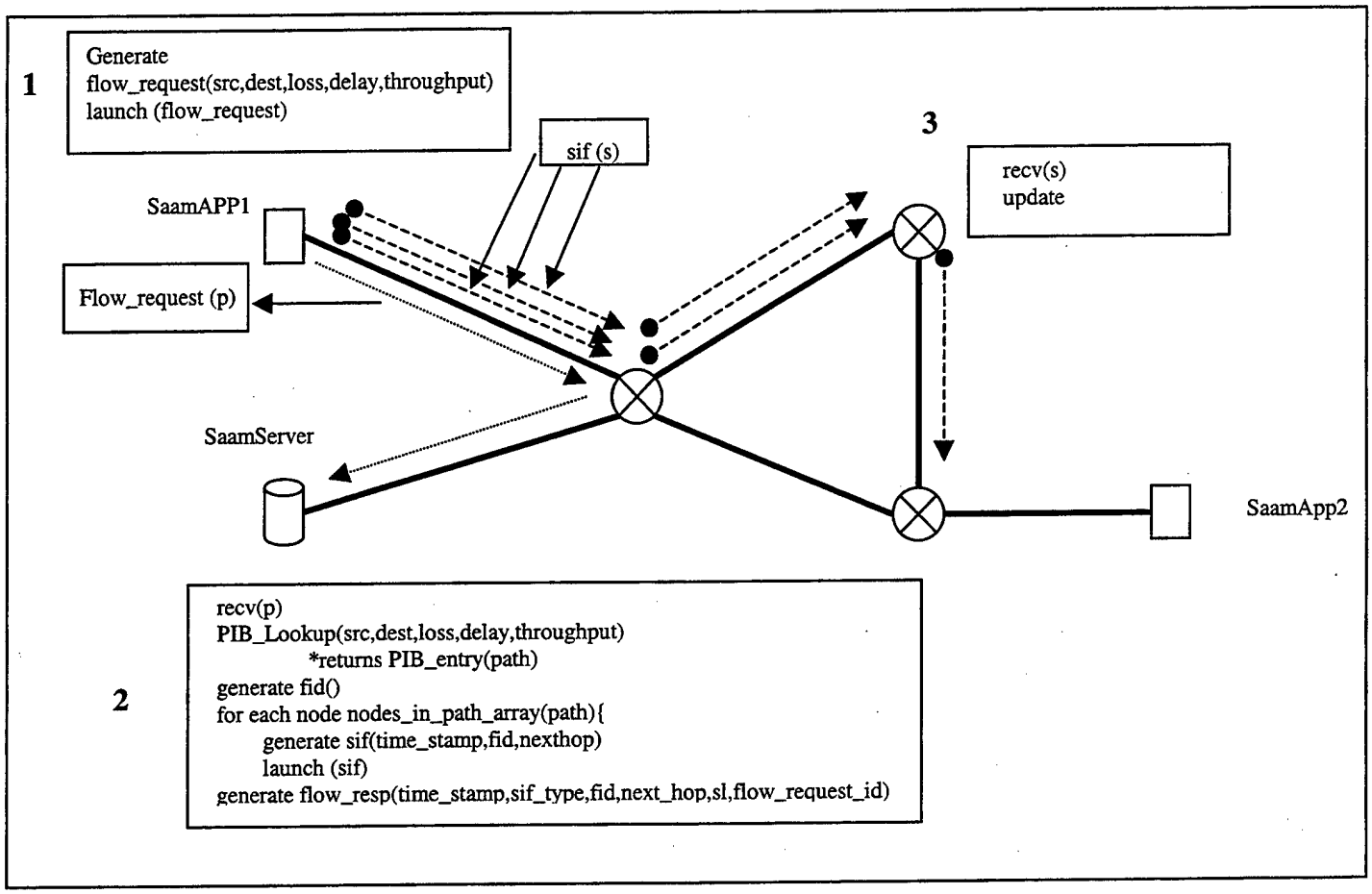


Figure 6. SAAM Execution Sequence Starting from Flow Request

2. SAAM EXECUTION SEQUENCE STARTING WITH AN LSA

The router code resides in "NSHOME/SaamRouter.cc". After the *recv* method is called the router accesses the SIF header and uses the *fid, next_hop* and *sl* fields to update its Flow Routing Table.

Figure 7 shows the SAAM execution sequence which is started when a router sends an LSA message to the server. The code which generates and sends an LSA can be found in "NSHOME/SaamRouter.cc". The router must be able to monitor all of the service level pipes on each of its associated links.[3] When a significant change in the QoS parameters occurs on a service level pipe the router will generate an LSA. The link is identified by the address of the origin router and the endpoint router that the link is attached to. The router then sends the LSA to the server.

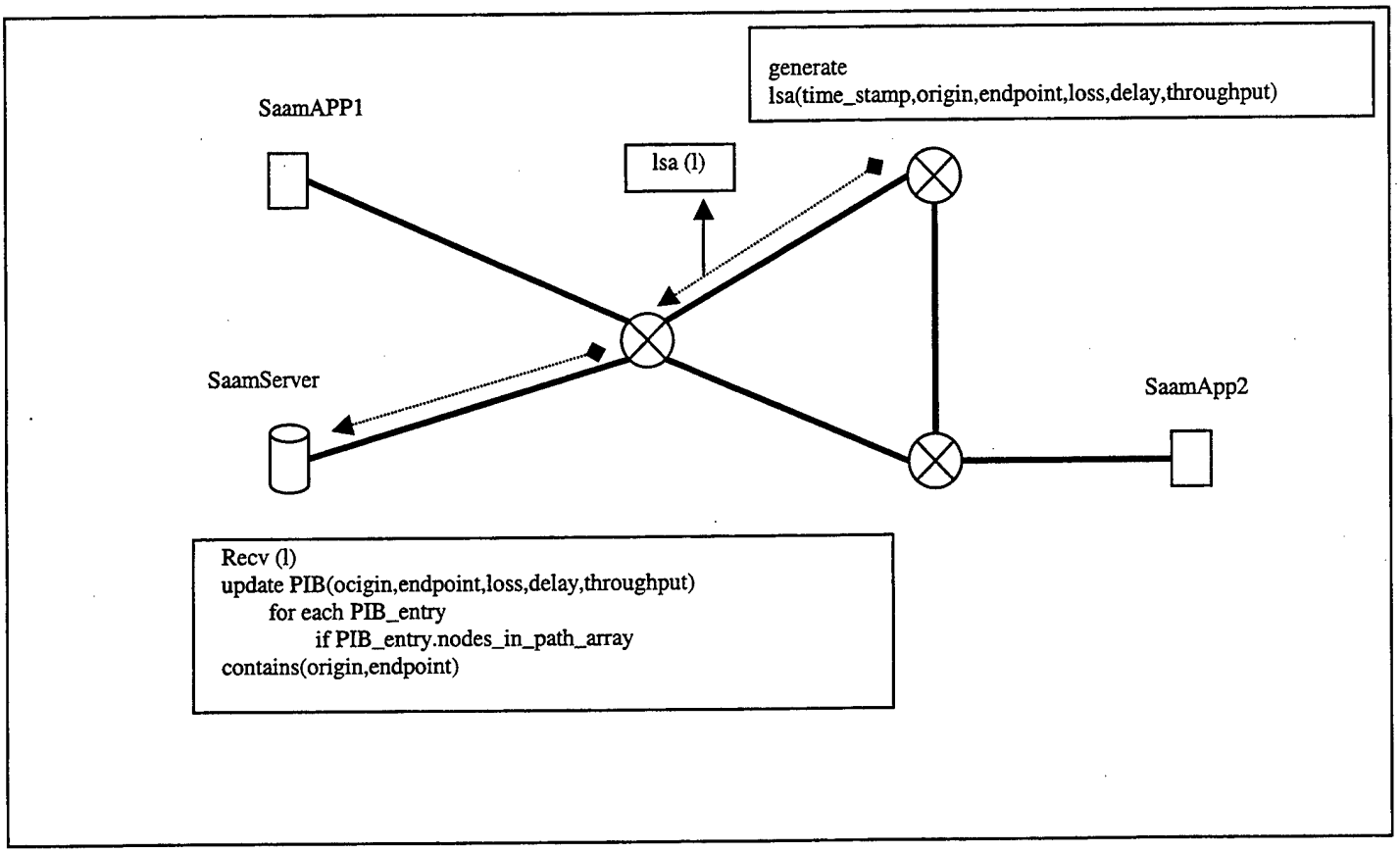


Figure 7. SAAM Execution Sequence Starting from Sending of LSA

Upon receipt of the LSA the server accesses the LSA header information. Using the *origin* and *endpoint* fields it indexes those *PIB_entry*'s which contains those routers in its *nodes_in_path_array* and updates the *loss*, *delay* and *throughput* fields.

[3] The current version of this model does not contain this functionality

In the creation of this model several NS source files had to be modified. The "NSHOME/packet.h" file contains the enumerated type *PTYPE*. *PTYPE* contains the names of all packet types that can be used in NS. If a new packet type is created when building a model it must be included as an enumerated member of *PTYPE*. The new packet types created in this model are PT_FLOW_RQST, PT_SIF and PT_LSA. All of these packet types had to be added to the enumerated type *PTYPE*. These packet types are used in the instantiation of an agent. When the agent's constructor is called the packet type that agent uses is sent as an argument to the parent object's constructor. The parent object for all agents is the class *Agent*.

The "Makefile" also had to be modified. When a new object is inserted to the model by adding source code files, the "Makefile" must be adjusted so that it knows to compile the additional files. The new file name with the extension replaced by a ".o" must be inserted into the *OBJ* portion of the "Makefile". This signals the compiler to source the new code and compile the object.

The new files created for this model were; "SaamApp.h","SaamApp.cc", "SaamRouter.h", SaamRouter.cc", SaamServer.h" and "SaamServer.cc". Each pair of new file required a new entry into the "Makefile".

3. NS RUNTIME INITIALIZATION

This section describes the initialization process the simulator goes through when a user runs the executable. The *main* program routine resides in HOME/tclAppinit.cc". The *main* program calls a *Tcl_main* program, which in turn calls *Tcl_AppInit*. *Tcl_AppInit* creates an interpreter process called *interp*. The interpreter process is an instance of an OTcl shell. The shell is created by a combination of function calls, *Tcl_Init* and *Otcl_Init*. *Tcl_Init* is located in "TCLHOME/tcl8.0/generic/tcl.h", and *Otcl_Init*is located in "*OTCLHOME/generic/otcl.h*".

Once the Tcl interpreter process has been started there are five steps in the Tcl_AppInit routine. The first step is to read in required resource files. This is done by the following command;

Tcl_SetVar(interp, "tcl_rcFileName", "~/.ns.tcl, TCL_GLOBAL_ONLY);

The next step is to name the shell instance. In all cases the shell gets the handle "ns". The code that does this is shown below;

Tcl::init(interp, "ns");

Once the shell has been named, the simulation must load all of the possible packet types that have been compiled and registered. The code to accomplish this is below;

e*t_ns_ptypes.load();*

The packet types registered in the file "NSHOME/packet.h" are turned into Tcl useable code through the "NSHOME/ptypes2tcl" file. They are put in the file "NSHOME/gen/ptypes.cc". In fact all NS files written in Tcl must be converted to C++ code in order to be recognized and accepted by the compiler. At run time during initialization the Otcl interpreter can access these items, both objects and commands, to instantiate required objects. This is also the method that allows Tcl code to be used by the simulation script. In this way all NS Tcl code is made available to the interpreter.

The next step is to load simulator object commands. This is accomplished in the same way that the packet types were loaded.

et_ns_lib.load();

Finally the *Tcl_AppInit* function loads all of the remaining miscellaneous commands;

Init_misc();

These miscellaneous commands include thing like random number generators and unit conversion functions.

Once these five steps are complete the the *main* routine contimues to run and execution never returns to it unless an error occurs. Subprograms run in the background which listen and await instructions and provide the insightful command prompt, *"%"*.

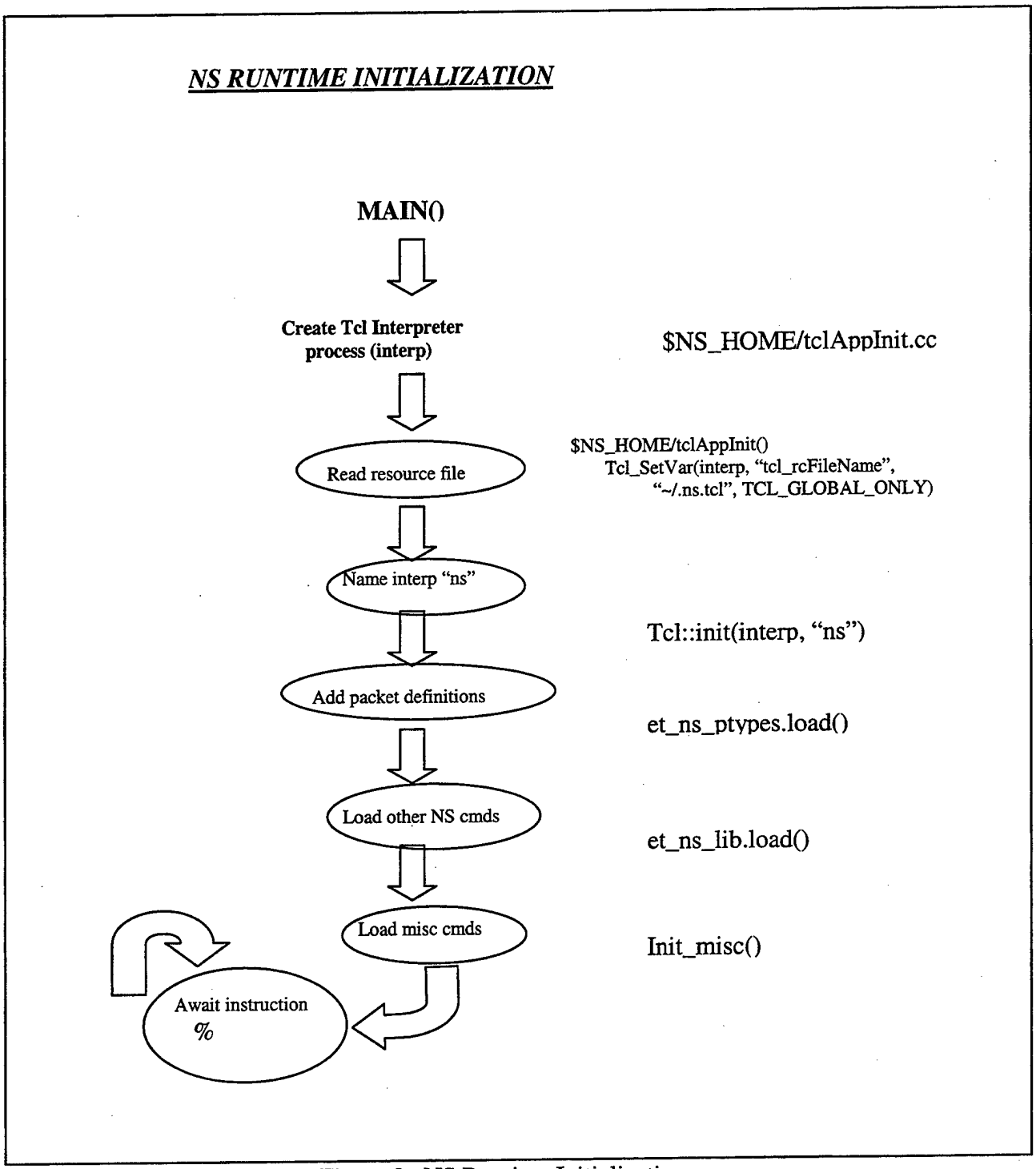


Figure 8. NS Runtime Initialization

APPENDIX C. [SERVER SOURCE CODE]

```
File name:      saamserver.h
 * Author:      Brian Tiefert
 * Date:        as of 12 Aug 1999
 * abilities:   able to create new saamserver through Tcl script
 *                      PIB definition
 *
 */

#ifndef ns_saamserver_h
#define ns_saamserver_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "address.h"
#include "ip.h"

struct hdr_sif{
        double time_stamp;
        int     sif_type;
        int     fid;
        nsaddr_t next_hop;
        int     sl;  //service level
        double flow_req_id;
  };

struct PIB_entry{
        nsaddr_t        src;
        nsaddr_t        dest;
        nsaddr_t        nodes_in_path_array[4];
        int             cost;
  };

//SaamServerAgent class definition

class SaamServerAgent : public Agent {
 public:
//member functions

//constructor
  SaamServerAgent();

  void initialize();
```

```
  int is_element(nsaddr_t array[], nsaddr_t router);
  int command(int argc, const char*const* argv);
//should override recv() from Agent but this is same method definition
  void recv(Packet*, Handler*);
 protected:
  int off_sif_;
  int off_lsa_;
  int off_flow_rqst_;
  nsaddr_t rtr_array[40] ;  //max of 40 routers allowed in simulation
  int router_index ;
  nsaddr_t app_array[2];   //2 apps allowed per simulation
  int app_index;

//for now set up a static PIB
//only 34 paths allowed
  PIB_entry PIB[34] ;
};

#endif
```

```
/*
 * File name:   saamserver.cc
 * Author:      Brian Tiefert
 * Date:        as of 17 Aug 1999
 * abilities:   Able to create a new saamserver through Tcl script,
 *              Able to send sif packets to router and recieve lsa's/flow_rqst
 *               creates a static PIB utilizing known topology
 *
 */


#include "saamrouter.h"
#include "saamserver.h"
#include "saamapp.h"
#include <iostream.h>

static class SaamServerHeaderClass : public PacketHeaderClass {
   public:
                SaamServerHeaderClass() : PacketHeaderClass("PacketHeader/Sif",
                                          sizeof(hdr_sif)) {}
} class_saamserverhdr;


//definition for SaamServerClass inherits from TclClass
static class SaamServerClass : public TclClass {
public:
//SaamServerClass constructor calls TclClass constructor using
//"Agent/SaamServer" as argument
  SaamServerClass() : TclClass("Agent/SaamServer") {}

  TclObject* create(int, const char*const*) {
//make a new SaamServerAgent by calling SaamServerAgent constructor
  return (new SaamServerAgent());
  }
} class_saamserver;

//constructor for SaamServerAgent calls constructor for Agent using PT_SIF as
//an argument
  SaamServerAgent::SaamServerAgent() : Agent(PT_SIF) {

  initialize();

//bind the compiled variables to the interpreted variables
  bind("packetSize_", &size_);
```

```
  bind("off_sif_", &off_sif_);
  bind("off_lsa_", &off_lsa_);

  cout<<"new SaamServer\n";

}

void SaamServerAgent::initialize()
{
        //first initialize the router array
        router_index = 0;
        for (int i=0; i<40; i++){
                rtr_array[i] = 0;}
        cout <<"rtr_array initialized\n";
        //next initialize application array
        app_index = 0;
        for  (int j=0; j<2; j++){
                app_array[j] = 9999;}
        cout <<"app_array initialized\n";

}

//is_element() array search function

int SaamServerAgent::is_element(nsaddr_t array[], nsaddr_t agent)
{
        int array_size;
        if (array == app_array) array_size = 2;
        else if (array == rtr_array) array_size = 40;
        for(int i = 0; i<array_size; i++){
                if (agent == array[i]){
                        return i;
                }
        }
        return -1;
}

/* The command function allows the developer to create methods
that can be accessed through the OTcl interpreter.  The function
takes a character string as an argument and compares it to the
developer created command name.  If a match is found the code
is executed
*/

int SaamServerAgent::command(int argc, const char*const* argv)
```

```
{
 if (argc == 2) {
  if (strcmp(argv[1], "launch") == 0) {
        //Create new packet
        Packet* pkt = allocpkt();
        //Access the Sif header for the new packet
        hdr_sif* hdr = (hdr_sif*)pkt->access(off_sif_);
        //Set the sif_type to LSA
        hdr->sif_type = 1;
        //Assign time_stamp
        hdr->time_stamp = Scheduler::instance().clock();
        //send the packet
        send(pkt, 0);
        //return TCL_OK, so the calling function knows that the
        //command has been processed
        return (TCL_OK);
        //Acknowledge the packet was sent
        cout<<"sif packet launched\n";


  }
  else if (strcmp(argv[1], "build_PIB") == 0) {
        cout<< "building PIB\n";
        // for now build a static PIB based on known topology
        nsaddr_t n0 = app_array[0];
        nsaddr_t n1 = this->addr_;
        nsaddr_t n2 = rtr_array[0];
        nsaddr_t n3 = rtr_array[1];
        nsaddr_t n4 = rtr_array[2];
        //n0->n1
        PIB[0].src = n0;
        PIB[0].dest= n1;
        PIB[0].nodes_in_path_array[0] = n0;
        PIB[0].nodes_in_path_array[1] = n2;
        PIB[0].nodes_in_path_array[2] = n1;
        PIB[0].cost = 2;
        //n0->n2
        PIB[1].src = n0;
        PIB[1].dest= n2;
        PIB[1].nodes_in_path_array[0] = n0;
        PIB[1].nodes_in_path_array[1] = n2;
        PIB[1].cost = 1;
        //n0->n3
        PIB[2].src = n0;
        PIB[2].dest= n3;
```

```
PIB[2].nodes_in_path_array[0] = n0;
PIB[2].nodes_in_path_array[1] = n2;
PIB[2].nodes_in_path_array[2] = n3;
PIB[2].cost = 3;
//n0->n3
PIB[3].src = n0;
PIB[3].dest= n3;
PIB[3].nodes_in_path_array[0] = n0;
PIB[3].nodes_in_path_array[1] = n2;
PIB[3].nodes_in_path_array[2] = n4;
PIB[3].nodes_in_path_array[3] = n3;
PIB[3].cost = 6;
//n0->n4
PIB[4].src = n0;
PIB[4].dest= n4;
PIB[4].nodes_in_path_array[0] = n0;
PIB[4].nodes_in_path_array[1] = n2;
PIB[4].nodes_in_path_array[2] = n4;
PIB[4].cost = 5;
//n0->n4
PIB[5].src = n0;
PIB[5].dest= n4;
PIB[5].nodes_in_path_array[0] = n0;
PIB[5].nodes_in_path_array[1] = n2;
PIB[5].nodes_in_path_array[2] = n3;
PIB[5].nodes_in_path_array[3] = n4;
PIB[5].cost = 4;
//n1->n0
PIB[6].src = n1;
PIB[6].dest= n0;
PIB[6].nodes_in_path_array[0] = n1;
PIB[6].nodes_in_path_array[1] = n2;
PIB[6].nodes_in_path_array[2] = n0;
PIB[6].cost = 2;
//n1->n2
PIB[7].src = n1;
PIB[7].dest= n2;
PIB[7].nodes_in_path_array[0] = n1;
PIB[7].nodes_in_path_array[1] = n2;
PIB[7].cost = 1;
//n1->n3
PIB[8].src = n1;
PIB[8].dest= n3;
PIB[8].nodes_in_path_array[0] = n1;
PIB[8].nodes_in_path_array[1] = n2;
```

```
PIB[8].nodes_in_path_array[2] = n3;
PIB[8].cost = 3;
//n1->n3
PIB[9].src = n1;
PIB[9].dest= n3;
PIB[9].nodes_in_path_array[0] = n1;
PIB[9].nodes_in_path_array[1] = n2;
PIB[9].nodes_in_path_array[2] = n4;
PIB[9].nodes_in_path_array[3] = n3;
PIB[9].cost = 6;
//n1->n4
PIB[10].src = n1;
PIB[10].dest= n4;
PIB[10].nodes_in_path_array[0] = n1;
PIB[10].nodes_in_path_array[1] = n2;
PIB[10].nodes_in_path_array[2] = n4;
PIB[10].cost = 5;
//n1->n4
PIB[11].src = n1;
PIB[11].dest= n4;
PIB[11].nodes_in_path_array[0] = n1;
PIB[11].nodes_in_path_array[1] = n2;
PIB[11].nodes_in_path_array[2] = n3;
PIB[11].nodes_in_path_array[3] = n4;
PIB[11].cost = 4;
//n2->n0
PIB[12].src = n2;
PIB[12].dest= n0;
PIB[12].nodes_in_path_array[0] = n2;
PIB[12].nodes_in_path_array[1] = n0;
PIB[12].cost = 1;
//n2->n1
PIB[13].src = n2;
PIB[13].dest= n1;
PIB[13].nodes_in_path_array[0] = n2;
PIB[13].nodes_in_path_array[1] = n1;
PIB[13].cost = 1;
//n2->n3
PIB[14].src = n2;
PIB[14].dest= n3;
PIB[14].nodes_in_path_array[0] = n2;
PIB[14].nodes_in_path_array[1] = n3;
PIB[14].cost = 2;
//n2->n3
PIB[15].src = n2;
```

```
PIB[15].dest= n3;
PIB[15].nodes_in_path_array[0] = n2;
PIB[15].nodes_in_path_array[1] = n4;
PIB[15].nodes_in_path_array[2] = n3;
PIB[15].cost = 5;
//n2->n4
PIB[16].src = n2;
PIB[16].dest= n4;
PIB[16].nodes_in_path_array[0] = n2;
PIB[16].nodes_in_path_array[1] = n4;
PIB[16].cost = 4;
//n2->n4
PIB[17].src = n2;
PIB[17].dest= n4;
PIB[17].nodes_in_path_array[0] = n2;
PIB[17].nodes_in_path_array[1] = n3;
PIB[17].nodes_in_path_array[2] = n4;
PIB[17].cost = 3;
//n3->n0
PIB[18].src = n3;
PIB[18].dest= n0;
PIB[18].nodes_in_path_array[0] = n3;
PIB[18].nodes_in_path_array[1] = n2;
PIB[18].nodes_in_path_array[2] = n0;
PIB[18].cost = 3;
//n3->n0
PIB[19].src = n3;
PIB[19].dest= n0;
PIB[19].nodes_in_path_array[0] = n3;
PIB[19].nodes_in_path_array[1] = n4;
PIB[19].nodes_in_path_array[2] = n2;
PIB[19].nodes_in_path_array[3] = n0;
PIB[19].cost = 6;
//n3->n1
PIB[20].src = n3;
PIB[20].dest= n1;
PIB[20].nodes_in_path_array[0] = n3;
PIB[20].nodes_in_path_array[1] = n2;
PIB[20].nodes_in_path_array[2] = n1;
PIB[20].cost = 3;
//n3->n1
PIB[21].src = n3;
PIB[21].dest= n1;
PIB[21].nodes_in_path_array[0] = n3;
PIB[21].nodes_in_path_array[1] = n4;
```

```
PIB[21].nodes_in_path_array[2] = n2;
PIB[21].nodes_in_path_array[3] = n1;
PIB[21].cost = 6;
//n3->n2
PIB[22].src = n3;
PIB[22].dest= n2;
PIB[22].nodes_in_path_array[0] = n3;
PIB[22].nodes_in_path_array[1] = n2;
PIB[22].cost = 2;
//n3->n2
PIB[23].src = n3;
PIB[23].dest= n2;
PIB[23].nodes_in_path_array[0] = n3;
PIB[23].nodes_in_path_array[1] = n4;
PIB[23].nodes_in_path_array[2] = n2;
PIB[23].cost = 5;
//n3->n4
PIB[24].src = n3;
PIB[24].dest= n4;
PIB[24].nodes_in_path_array[0] = n3;
PIB[24].nodes_in_path_array[1] = n4;
PIB[24].cost = 1;
//n3->n4
PIB[25].src = n3;
PIB[25].dest= n4;
PIB[25].nodes_in_path_array[0] = n3;
PIB[25].nodes_in_path_array[1] = n2;
PIB[25].nodes_in_path_array[2] = n4;
PIB[25].cost = 6;
//n4->n0
PIB[26].src = n4;
PIB[26].dest= n0;
PIB[26].nodes_in_path_array[0] = n4;
PIB[26].nodes_in_path_array[1] = n2;
PIB[26].nodes_in_path_array[2] = n0;
PIB[26].cost = 5;
//n4->n0
PIB[27].src = n4;
PIB[27].dest= n0;
PIB[27].nodes_in_path_array[0] = n4;
PIB[27].nodes_in_path_array[1] = n3;
PIB[27].nodes_in_path_array[2] = n2;
PIB[27].nodes_in_path_array[3] = n0;
PIB[27].cost = 4;
//n4->n1
```

```
        PIB[28].src = n4;
        PIB[28].dest= n1;
        PIB[28].nodes_in_path_array[0] = n4;
        PIB[28].nodes_in_path_array[1] = n2;
        PIB[28].nodes_in_path_array[2] = n1;
        PIB[28].cost = 5;
        //n4->n1
        PIB[29].src = n4;
        PIB[29].dest= n1;
        PIB[29].nodes_in_path_array[0] = n4;
        PIB[29].nodes_in_path_array[1] = n3;
        PIB[29].nodes_in_path_array[2] = n2;
        PIB[29].nodes_in_path_array[3] = n1;
        PIB[29].cost = 4;
        //n4->n2
        PIB[30].src = n4;
        PIB[30].dest= n2;
        PIB[30].nodes_in_path_array[0] = n4;
        PIB[30].nodes_in_path_array[1] = n2;
        PIB[30].cost = 4;
        //n4->n2
        PIB[31].src = n4;
        PIB[31].dest= n2;
        PIB[31].nodes_in_path_array[0] = n4;
        PIB[31].nodes_in_path_array[1] = n3;
        PIB[31].nodes_in_path_array[2] = n2;
        PIB[31].cost = 3;
        //n4->n3
        PIB[32].src = n4;
        PIB[32].dest= n3;
        PIB[32].nodes_in_path_array[0] = n4;
        PIB[32].nodes_in_path_array[1] = n3;
        PIB[32].cost = 1;
        //n4->n3
        PIB[33].src = n4;
        PIB[33].dest= n3;
        PIB[33].nodes_in_path_array[0] = n4;
        PIB[33].nodes_in_path_array[1] = n2;
        PIB[32].nodes_in_path_array[2] = n3;
        PIB[33].cost = 5;

        return (TCL_OK);
    }
}
```

```
// If the command hasn't been processed by SaamServerAgent()::command,
// call the command() function for the base class
 return (Agent::command(argc, argv));
}
/* The receive method tells the agent how to handle incoming
packets addressed to the agent*/

void SaamServerAgent::recv(Packet* pkt, Handler*)
{
 //Access the IP header for the received Packet
 hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
 //Access the common header for the received packet
 hdr_cmn* hdrcmn = (hdr_cmn*)pkt->access(off_cmn_);
 if (hdrcmn->ptype_ == PT_LSA){
        //cout<<"lsa packet recieved\n";
        //Access the lsa_hdr
        hdr_lsa* hdrlsa = (hdr_lsa*)pkt->access(off_lsa_);
   //cout<<"time of lsa packet recv'd is "<<Scheduler::instance().clock()<<endl;
        // determine if src router is     element of rtr_array
        int index = is_element(rtr_array, hdrip->src_);
        if (index == -1) {
                cout<<"router not in array so add\n";
                rtr_array[router_index] = hdrip->src_ ;
                nsaddr_t newrtr = rtr_array[router_index];
                //cout<<"router added was "<< newrtr <<endl;
                router_index ++ ;
        } //endif

 }

 else if (hdrcmn->ptype_ == PT_FLOW_RQST){
        //access the flow_rqst header
     hdr_flow_rqst* flowRqstHdr=(hdr_flow_rqst*)pkt->access(off_flow_rqst_);
        //double tmstamp =flowRqstHdr->time_stamp;
        cout << "Flow request received at server\n";
   //cout<<"Flow request was received at:"<<Scheduler::instance().clock()<<endl;

        // determine if app is element of app_array
        int index = is_element(app_array, hdrip->src_);
        if (index == -1) {
                cout<<"app not in array so add\n";
                app_array[app_index] = hdrip->src_ ;
                nsaddr_t newrtr = app_array[app_index];
                //cout<<"app added was "<< newrtr <<endl;
                app_index ++ ;
```

```
        } //endif

        // send flow_resp
        Packet* resppkt = allocpkt();
        //Access the Sif header for the new packet
        hdr_sif* hdr = (hdr_sif*)resppkt->access(off_sif_);
        // need to fill in flow_resp info here
        hdr->sif_type = 3;
        hdr->time_stamp = Scheduler::instance().clock();
        send(resppkt, 0);
        //cout<<"flow_resp packet launched at: " << hdr->time_stamp <<endl;
        // send frt_add to router
        // first need to change target_
        // need to find routers addresses
        for (int i=0; i<40; i++){
                nsaddr_t rtr = rtr_array[i];
                if (rtr != 0){
                this->dst_ = rtr;
                Packet* frtpkt = allocpkt();
                //Access the Sif header for the new packet
                hdr_sif* resphdr = (hdr_sif*)frtpkt->access(off_sif_);
                // need to fill in flow_resp info here
                resphdr->sif_type = 1;
                resphdr->time_stamp = Scheduler::instance().clock();
                send(frtpkt, 0);
              //cout<<"frt_add packet launched at: " <<resphdr->time_stamp;
                //cout<< "to " << rtr <<endl;
        }

        }
 }

 else {
        //A packet was received.  Use tcl.eval to call the Tcl interpreter
        // with the ping results.
        char out[100];

  //sprintf(out, "%s recv %d %3.1f", name(),
        //hdrip->src_>> Address::instance().NodeShift_[1],
     //(Scheduler::instance().clock()-flowRqstHdr->time_stamp) * 1000);
  Tcl& tcl = Tcl::instance();
  tcl.eval(out);
//Discard the packet
  Packet::free(pkt);
  }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. [ROUTER SOURCE CODE]

```
/*
 * File name:   saamrouter.h
 * Author:      Brian Tiefert
 * Date:        as of 12 Aug 1999

 */

#ifndef ns_saamrouter_h
#define ns_saamrouter_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "address.h"
#include "ip.h"

struct hdr_lsa{
        double time_stamp;
        nsaddr_t origin;
        nsaddr_t endpt;
        int     delay;
        int     loss;
        int     throughput;
  };

//SaamRouterAgent class definition

class SaamRouterAgent : public Agent {
 public:
//member functions

//constructor
  SaamRouterAgent();
//
  int command(int argc, const char*const* argv);
//should override recv() from Agent but this is same method definition
  void recv(Packet*, Handler*);
 protected:
  int off_lsa_;
  int off_sif_;
};

#endif
```

```
/*
 * File name:   saamrouter.cc
 * Author:      Brian Tiefert
 * Date:        as of 12 Aug 1999
 * abilities:   Able to recv sif packets from server and send lsa's to server
 *
 */

#include "saamserver.h"
#include "saamrouter.h"
#include <iostream.h>
static class SaamRouterHeaderClass : public PacketHeaderClass {
public:
  SaamRouterHeaderClass() : PacketHeaderClass("PacketHeader/Lsa",
                                              sizeof(hdr_lsa)) {}
} class_saamrouterhdr;



//definition for SaamRouterClass inherits from TclClass
static class SaamRouterClass : public TclClass {
public:
//member methods

//SaamRouterClass constructor calls TclClass constructor using
//"Agent/SaamRouter" as argument
  SaamRouterClass() : TclClass("Agent/SaamRouter") {}
 TclObject* create(int, const char*const*) {

//make a new SaamRouterAgent by calling SaamRouterAgent constructor
  return (new SaamRouterAgent());

  }
} class_saamrouter;

//constructor for SaamRouterAgent calls constructor for Agent using PT_LSA as
//an argument
        SaamRouterAgent::SaamRouterAgent() : Agent(PT_LSA) {
//bind the compiled variables to the interpreted variables
  bind("packetSize_", &size_);
  bind("off_lsa_", &off_lsa_);
  bind("off_sif_", &off_sif_);
  cout<<"new SaamRouter\n";
}
```

```
int SaamRouterAgent::command(int argc, const char*const* argv)
{
 if (argc == 2) {
  if (strcmp(argv[1], "launch") == 0) {
        //Create new packet
        Packet* pkt = allocpkt();
        //Access the Lsa header for the new packet
        hdr_lsa* hdr = (hdr_lsa*)pkt->access(off_lsa_);
        // Set the 'ret' field to 0, so the receiving node knows
        // that it has to generate an echo packet
        //hdr->sif_type = 1;
        hdr->time_stamp = Scheduler::instance().clock();
        send(pkt, 0);
        //return TCL_OK, so the calling function knows that the
        //command has been processed
        cout<<"lsa packet launched\n";
        return (TCL_OK);
  }
  else if (strcmp(argv[1], "initialize") == 0) {
        //Create new packet
        Packet* pkt = allocpkt();
        //Access the Lsa header for the new packet
        hdr_lsa* hdr = (hdr_lsa*)pkt->access(off_lsa_);
        hdr->time_stamp = Scheduler::instance().clock();
        send(pkt, 0);
        //return TCL_OK, so the calling function knows that the
        //command has been processed
        cout<<"lsa packet launched\n";
        return (TCL_OK);
  }

}

// If the command hasn't been processed by SaamRouterAgent()::command,
// call the command() function for the base class
 return (Agent::command(argc, argv));
}

void SaamRouterAgent::recv(Packet* pkt, Handler*)
{
 //Access the IP header for the received Packet
 hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
  // if pkt is of type lsa then access hdr_lsa
 // Access the Sif header for the received packet
```

```
 hdr_sif* hdr = (hdr_sif*)pkt->access(off_sif_);
 if (hdr->sif_type == 1){

        cout<<"frt_add packet recieved\n";
        //Send an echo.  First save the old packet's send_time
        double tmstamp = hdr->time_stamp;
        cout << "time of frt_add sent packet is " << tmstamp << endl;
        //Discard the packet
        //Packet::free(pkt);

//send packet to rtprotoSAAM or RouteLogic
        //Create new packet
        Packet* pktret = allocpkt();
        //Access the Ping header for the new packet
        hdr_lsa* hdrret = (hdr_lsa*)pktret->access(off_lsa_);
        //Set the time_stamp field to the correct value
        hdrret->time_stamp = Scheduler::instance().clock();
        //send packet to dst_
        send(pktret, 0);
        }
 else if (hdr->sif_type == 2) cout<<"frt_del packet received\n";

 else if (hdr->sif_type == 3) cout<<"flow_resp packet received\n";

else {
  //A packet was received.  Use tcl.eval to call the Tcl interpreter
  // with the ping results.
  char out[100];

  sprintf(out, "%s recv %d %3.1f", name(),
        hdrip->src_>> Address::instance().NodeShift_[1],
     (Scheduler::instance().clock()-hdr->time_stamp) * 1000);
  Tcl& tcl = Tcl::instance();
  tcl.eval(out);
//Discard the packet
 Packet::free(pkt);
 }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. [MODIFIED SOURCE CODE]

```
/* -*-	Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    This product includes software developed by the Computer Systems
 *    Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * @(#) $Header: /usr/src/mash/repository/vint/ns-2/packet.h,v 1.59 1999/03/13 03:52:59 haoboy Exp $ (LBL)
 */

#ifndef ns_packet_h
#define ns_packet_h

#include <string.h>
#include <assert.h>

#include "config.h"
#include "scheduler.h"
#include "object.h"
#include "list.h"
#include "packet-stamp.h"


#define RT_PORT		255	/* port that all route msgs are sent to */
```

```
#define HDR_CMN(p)      ((struct hdr_cmn*)(p)->access(hdr_cmn::offset_))
#define HDR_ARP(p)      ((struct hdr_arp*)(p)->access(off_arp_))
#define HDR_MAC(p)      ((struct hdr_mac*)(p)->access(hdr_mac::offset_))
#define HDR_MAC802_11(p) ((struct hdr_mac802_11*)(p)->access(hdr_mac::offset_))
#define HDR_LL(p)       ((struct hdr_ll*)(p)->access(hdr_ll::offset_))
#define HDR_IP(p)       ((struct hdr_ip*)(p)->access(hdr_ip::offset_))
#define HDR_SAAM(p)          ((struct hdr_saam*)(p)->access(hdr_saam::offset_))
#define HDR_SIF(p)      ((struct hdr_sif*)(p)->access(hdr_sif::offset_))
#define HDR_RTP(p)      ((struct hdr_rtp*)(p)->access(hdr_rtp::offset_))
#define HDR_TCP(p)      ((struct hdr_tcp*)(p)->access(hdr_tcp::offset_))

enum packet_t {
        PT_TCP,
        PT_UDP,
        PT_CBR,
        PT_AUDIO,
        PT_VIDEO,
        PT_ACK,
        PT_START,
        PT_STOP,
        PT_PRUNE,
        PT_GRAFT,
        PT_GRAFTACK,
        PT_JOIN,
        PT_ASSERT,
        PT_MESSAGE,
        PT_RTCP,
        PT_RTP,
        PT_RTPROTO_DV,
        PT_RTPROTO_SAAM,
        PT_CtrMcast_Encap,
        PT_CtrMcast_Decap,
        PT_SRM,

        /* simple signalling messages */
        PT_REQUEST,
        PT_ACCEPT,
        PT_CONFIRM,
        PT_TEARDOWN,
        PT_LIVE,        // packet from live network
        PT_REJECT,

        PT_TELNET,   // not needed: telnet use TCP
        PT_FTP,
        PT_PARETO,
        PT_EXP,
        PT_INVAL,
        PT_HTTP,

        /* new encapsulator */
```

```
        PT_ENCAPSULATED,
        PT_MFTP,

        /* CMU/Monarch's extnsions */
        PT_ARP,
        PT_MAC,
        PT_TORA,
        PT_DSR,
        PT_AODV,

        // RAP packets
        PT_RAP_DATA,
        PT_RAP_ACK,

        // insert new packet types here
        PT_FLOW_RQST, //new packet for flow requests
        PT_SIF,  // new packet for SaamServer
        PT_LSA,  // new packet for SaamRouter
        PT_SAAM, // new packet type for saam
        PT_ECHO,  // new packet type for ping requests
        PT_PING, // new packet type for ping requests
        PT_NTYPE // This MUST be the LAST one
};

class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
                name_[PT_UDP]= "udp";
                name_[PT_CBR]= "cbr";
                name_[PT_AUDIO]= "audio";
                name_[PT_VIDEO]= "video";
                name_[PT_ACK]= "ack";
                name_[PT_START]= "start";
                name_[PT_STOP]= "stop";
                name_[PT_PRUNE]= "prune";
                name_[PT_GRAFT]= "graft";
                name_[PT_GRAFTACK]= "graftAck";
                name_[PT_JOIN]= "join";
                name_[PT_ASSERT]= "assert";
                name_[PT_MESSAGE]= "message";
                name_[PT_RTCP]= "rtcp";
                name_[PT_RTP]= "rtp";
                name_[PT_RTPROTO_DV]= "rtProtoDV";
                name_[PT_RTPROTO_SAAM]= "rtProtoSAAM";
                name_[PT_CtrMcast_Encap]= "CtrMcast_Encap";
                name_[PT_CtrMcast_Decap]= "CtrMcast_Decap";
                name_[PT_SRM]= "SRM";

                name_[PT_REQUEST]= "sa_req";
```

```
                name_[PT_ACCEPT]= "sa_accept";
                name_[PT_CONFIRM]= "sa_conf";
                name_[PT_TEARDOWN]= "sa_teardown";
                name_[PT_LIVE]= "live";
                name_[PT_REJECT]= "sa_reject";

                name_[PT_TELNET]= "telnet";
                name_[PT_FTP]= "ftp";
                name_[PT_PARETO]= "pareto";
                name_[PT_EXP]= "exp";
                name_[PT_INVAL]= "httpInval";
                name_[PT_HTTP]= "http";
                name_[PT_ENCAPSULATED]= "encap";
                name_[PT_MFTP]= "mftp";
                name_[PT_ARP]= "ARP";
                name_[PT_MAC]= "MAC";
                name_[PT_TORA]= "TORA";
                name_[PT_DSR]= "DSR";
                name_[PT_AODV]= "AODV";

                name_[PT_RAP_DATA] = "rap_data";
                name_[PT_RAP_ACK] = "rap_ack";
                name_[PT_FLOW_RQST] = "flow_rqst";
                name_[PT_SIF] = "Sif";
                name_[PT_LSA] = "Lsa";
                name_[PT_SAAM] = "Saam";
                name_[PT_ECHO] = "Echo";
                name_[PT_PING] = "Ping";
                name_[PT_NTYPE]= "undefined";
        }
        const char* name(packet_t p) const {
                if ( p <= PT_NTYPE ) return name_[p];
                return 0;
        }
        static bool data_packet(packet_t type) {
                return ( (type) == PT_TCP || \
                        (type) == PT_TELNET || \
                        (type) == PT_CBR || \
                        (type) == PT_AUDIO || \
                        (type) == PT_VIDEO || \
                        (type) == PT_ACK  );
        }
private:
        static char* name_[PT_NTYPE+1];
};
extern p_info packet_info; /* map PT_* to string name */
//extern char* p_info::name_[];

#define OFFSET(type, field)     ((int) &((type *)0)->field)
```

```
//Monarch ext
typedef void (*FailureCallback)(Packet *,void *);
//
class Packet : public Event {
private:
        unsigned char* bits_;   // header bits
        unsigned char* data_;   // variable size buffer for 'data'
        unsigned int datalen_;  // length of variable size buffer
    //void init();          // initialize pkts getting freed.
        bool fflag_;
protected:
        static Packet* free_;   // packet free list
public:
        Packet* next_;          // for queues and the free list
        static int hdrlen_;
        Packet() : bits_(0), datalen_(0), next_(0) { }
        unsigned char* const bits() { return (bits_); }
        Packet* copy() const;
        static Packet* alloc();
        static Packet* alloc(int);
        inline void allocdata(int);
        static void free(Packet*);
        inline unsigned char* access(int off) const {
                if (off < 0)
                        abort();
                return (&bits_[off]);
        }
        inline unsigned char* accessdata() const { return data_; }
        inline int datalen() const { return datalen_; }

        //Monarch extn
        // the pkt stamp carries all info about how/where the pkt
    // was sent needed for a receiver to determine if it correctly
    // receives the pkt

    PacketStamp     txinfo_;

        //monarch extns end;

};

/*
 * static constant associations between interface special (negative)
 * values and their c-string representations that are used from tcl
 */
class iface_literal {
public:
        enum iface_constant {
                UNKN_IFACE= -1, /* iface value for locally originated packets
                                */
```

```
                ANY_IFACE= -2  /* hashnode with iif == ANY_IFACE_
                               * matches any pkt iface (imported from TCL);
                               * this value should be different from hdr_cmn::UNKN_IFACE
                               * from packet.h
                               */
        };
        iface_literal(const iface_constant i, const char * const n) :
                value_(i), name_(n) {}
        inline int value() const { return value_; }
        inline const char * const name() const { return name_; }
private:
        const iface_constant value_;
        const char * const name_; /* strings used in TCL to access those special values */
};

static const iface_literal UNKN_IFACE(iface_literal::UNKN_IFACE, "?");
static const iface_literal ANY_IFACE(iface_literal::ANY_IFACE, "*");


struct hdr_cmn {
        packet_t ptype_;                // packet type (see above)
        int      size_;         // simulated packet size
        int      uid_;          // unique id
        int      error_;        // error flag
        double   ts_;           // timestamp: for q-delay measurement
        int      iface_;        // receiving interface (label)
        int      direction_;    // direction: 0=none, 1=up, -1=down
        int      ref_count_;    // free the pkt until count to 0

        //Monarch extn begins
        nsaddr_t next_hop_;     // next hop for this packet
        int     addr_type_;   // type of next_hop_ addr
#define AF_NONE 0
#define AF_ILINK 1
#define AF_INET 2

    // called if pkt can't obtain media or isn't ack'd. not called if
    // droped by a queue
    FailureCallback xmit_failure_;
    void *xmit_failure_data_;

    /*
     * MONARCH wants to know if the MAC layer is passing this back because
     * it could not get the RTS through or because it did not receive
     * an ACK.
     */
    int    xmit_reason_;
#define XMIT_REASON_RTS 0x01
#define XMIT_REASON_ACK 0x02
```

```
    // filled in by GOD on first transmission, used for trace analysis
    int num_forwards_;          // how many times this pkt was forwarded
    int opt_num_forwards_;  // optimal #forwards
// Monarch extn ends;

        static int offset_;         // offset for this header
        inline static int& offset() { return offset_; }
        inline static hdr_cmn* access(Packet* p) {
                return (hdr_cmn*) p->access(offset_);
        }

    /* per-field member functions */
        inline packet_t& ptype() { return (ptype_); }
        inline int& size() { return (size_); }
        inline int& uid() { return (uid_); }
        inline int& error() { return error_; }
        inline double& timestamp() { return (ts_); }
        inline int& iface() { return (iface_); }
        inline int& direction() { return (direction_); }
        inline int& ref_count() { return (ref_count_); }
        // monarch_begin
        inline nsaddr_t& next_hop() { return (next_hop_); }
        inline int& addr_type() { return (addr_type_); }
        inline int& num_forwards() { return (num_forwards_); }
        inline int& opt_num_forwards() { return (opt_num_forwards_); }
    //monarch_end
};


class PacketHeaderClass : public TclClass {
protected:
        PacketHeaderClass(const char* classname, int hdrsize);
        virtual int method(int argc, const char*const* argv);
        void field_offset(const char* fieldname, int offset);
        inline void bind_offset(int* off) { offset_ = off; }
        inline void offset(int* off) {offset_= off;}
        int hdrlen_;                // # of bytes for this header
        int* offset_;               // offset for this header
public:
        virtual void bind();
        virtual void export_offsets();
        TclObject* create(int argc, const char*const* argv);
};


inline Packet* Packet::alloc()
{
        Packet* p = free_;
        if (p != 0) {
                assert(p->fflag_ == FALSE);
```

```
                free_ = p->next_;
                if (p->datalen_) {
                        delete[] p->data_;
                        // p->data_ = 0;
                        p->datalen_ = 0;
                }
                p->uid_ = 0;
                p->time_ = 0;
        }
        else {
                p = new Packet;
                p->bits_ = new unsigned char[hdrlen_];
                if (p == 0 || p->bits_ == 0)
                        abort();
//              p->data_ = 0;
//              p->datalen_ = 0;
                bzero(p->bits_, hdrlen_);
        }
        p->fflag_ = TRUE;
        p->next_ = 0;
        return (p);
}

/* allocate a packet with an n byte data buffer */

inline Packet* Packet::alloc(int n)
{
        Packet* p = alloc();
        if (n > 0)
                p->allocdata(n);
        return (p);
}

/* allocate an n byte data buffer to an existing packet */

inline void Packet::allocdata(int n)
{
        datalen_ = n;
        data_ = new unsigned char[n];
        if (data_ == 0)
                abort();

}

inline void Packet::free(Packet* p)
{
        int off_cmn_ = hdr_cmn::offset_;
        hdr_cmn* ch = (hdr_cmn*)p->access(off_cmn_);
        if (p->fflag_) {
                if (ch->ref_count() == 0) {
```

```
                        /*
                         * A packet's uid may be < 0 (out of a event queue), or
                         * == 0 (newed but never gets into the event queue.
                         */
                        assert(p->uid_ <= 0);
                        p->next_ = free_;
                        free_ = p;
                        //init();
                        p->fflag_ = FALSE;
                } else {
                        ch->ref_count() = ch->ref_count() - 1;
                }
        }
}

inline Packet* Packet::copy() const
{
        Packet* p = alloc();
        memcpy(p->bits(), bits_, hdrlen_);
        if (datalen_) {
                p->datalen_ = datalen_;
                p->data_ = new unsigned char[datalen_];
                memcpy(p->data_, data_, datalen_);
        }
        p->txinfo_.init(&txinfo_);
        return (p);
}

#endif
```

```
#
# Copyright (c) 1997 by the University of Southern California
# All rights reserved.
#
# File Name:        route-proto.tcl
# Modified by:      Brian Tiefert
# Date:             as of 20 Sept 1999
#
# Permission to use, copy, modify, and distribute this software and its
# documentation in source and binary forms for non-commercial purposes
# and without fee is hereby granted, provided that the above copyright
# notice appear in all copies and that both the copyright notice and
# this permission notice appear in supporting documentation. and that
# any documentation, advertising materials, and other materials related
# to such distribution and use acknowledge that the software was
# developed by the University of Southern California, Information
# Sciences Institute. The name of the University may not be used to
# endorse or promote products derived from this software without
# specific prior written permission.
#
# THE UNIVERSITY OF SOUTHERN CALIFORNIA makes no representations about
# the suitability of this software for any purpose. THIS SOFTWARE IS
# PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES,
# INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
#
# Other copyrights might apply to parts of this software and are so
# noted when applicable.
#

#
# Maintainer: <kannan@isi.edu>.
#

# This file only contains the methods for dynamic routing
# Check ../lib/ns-route.tcl for the Simulator routing support
#

    set rtglibRNG [new RNG]
    $rtglibRNG seed 1

    Class rtObject

    rtObject set unreach_ -1
    rtObject set maxpref_  255

    rtObject proc init-all args {
       foreach node $args {
            if { [$node rtObject?] == "" } {
               set rtobj($node) [new rtObject $node]
```

```
        }
    }
    foreach node $args {  ;# XXX
        $rtobj($node) compute-routes
    }
}

rtObject instproc init node {
    $self next
    $self instvar ns_ nullAgent_
    $self instvar nextHop_ rtpref_ metric_ node_ rtVia_ rtProtos_

    set ns_ [Simulator instance]
    set nullAgent_ [$ns_ set nullAgent_]

    $node init-routing $self
    set node_ $node
    foreach dest [$ns_ all-nodes-list] {
        set nextHop_($dest) ""
        if {$node == $dest} {
            set rtpref_($dest) 0
            set metric_($dest) 0
            set rtVia_($dest) "Agent/rtProto/Local" ;# make dump happy
        } else {
            set rtpref_($dest) [$class set maxpref_]
            set metric_($dest) [$class set unreach_]
            set rtVia_($dest)   ""
            $node add-route [$dest id] $nullAgent_
        }
    }
    $self add-proto Direct $node
    $rtProtos_(Direct) compute-routes
#   $self compute-routes
}

rtObject instproc add-proto {proto node} {
    $self instvar ns_ rtProtos_
    set rtProtos_($proto) [new Agent/rtProto/$proto $node]
    $ns_ attach-agent $node $rtProtos_($proto)
    set rtProtos_($proto)
    # update nodes Protos_
    #$node updateProtos rtProtos_

}


rtObject instproc lookup dest {
    $self instvar nextHop_ node_
    if {![info exists nextHop_($dest)] || $nextHop_($dest) == ""} {
        return -1
```

```
    } else {
        return [[$nextHop_($dest) set toNode_] id]
    }
}

rtObject instproc compute-routes {} {
    # choose the best route to each destination from all protocols
    $self instvar ns_ node_ rtProtos_ nullAgent_
    $self instvar nextHop_ rtpref_ metric_ rtVia_
    set protos ""
    set changes 0
    foreach p [array names rtProtos_] {
        if [$rtProtos_($p) set rtsChanged_] {
            incr changes
            $rtProtos_($p) set rtsChanged_ 0
        }
        lappend protos $rtProtos_($p)
    }
    if !$changes return

    set changes 0
    foreach dst [$ns_ all-nodes-list] {
        if {$dst == $node_} continue
        set nh ""
        set pf [$class set maxpref_]
        set mt [$class set unreach_]
        set rv ""
        foreach p $protos {
            set pnh [$p set nextHop_($dst)]
            if { $pnh == "" } continue

            set ppf [$p set rtpref_($dst)]
            set pmt [$p set metric_($dst)]
            if {$ppf < $pf || ($ppf == $pf && $pmt < $mt) || $mt < 0} {
                set nh  $pnh
                set pf  $ppf
                set mt  $pmt
                set rv  $p
            }
        }
        if { $nh == "" } {
            # no route... delete any existing routes
            if { $nextHop_($dst) != "" } {
                $node_ delete-routes [$dst id] $nextHop_($dst) $nullAgent_
                set nextHop_($dst) $nh
                set rtpref_($dst)  $pf
                set metric_($dst)  $mt
                set rtVia_($dst)   $rv
                incr changes
            }
```

```
        } else {
          if { $rv == $rtVia_($dst) } {
                # Current protocol still has best route.
                # See if changed
                if { $nh != $nextHop_($dst) } {
                   $node_ delete-routes [$dst id] $nextHop_($dst) $nullAgent_
                   set nextHop_($dst) $nh
                   $node_ add-routes [$dst id] $nextHop_($dst)
                   incr changes
                }
                if { $mt != $metric_($dst) } {
                   set metric_($dst) $mt
                   incr changes
                }
                if { $pf != $rtpref_($dst) } {
                   set rtpref_($dst) $pf
                }
          } else {
                if { $rtVia_($dst) != "" } {
                   set nextHop_($dst) [$rtVia_($dst) set nextHop_($dst)]
                   set rtpref_($dst)  [$rtVia_($dst) set rtpref_($dst)]
                   set metric_($dst)  [$rtVia_($dst) set metric_($dst)]
                }
                if {$rtpref_($dst) != $pf || $metric_($dst) != $mt} {
                   # Then new prefs must be better, or
                   # new prefs are equal, and new metrics are lower
                   $node_ delete-routes [$dst id] $nextHop_($dst) $nullAgent_
                   set nextHop_($dst) $nh
                   set rtpref_($dst)  $pf
                   set metric_($dst)  $mt
                   set rtVia_($dst)   $rv
                   $node_ add-routes [$dst id] $nextHop_($dst)
                   incr changes
                }
          }
        }
}
foreach proto [array names rtProtos_] {
        $rtProtos_($proto) send-updates $changes
}
#
# XXX
# detailed multicast routing hooks must come here.
# My idea for the hook will be something like:
# set mrtObject [$node_ mrtObject?]
# if {$mrtObject != ""} {
#    $mrtObject recompute-mroutes $changes
# }
# $changes == 0         if only interfaces changed state.  Look at how
#                       Agent/rtProto/DV handles ifsUp_
```

```
    # $changes > 0        if new unicast routes were installed.
    #
    $self flag-multicast $changes
}

rtObject instproc flag-multicast changes {
    $self instvar node_
    $node_ notify-mcast $changes
}

rtObject instproc intf-changed {} {
    $self instvar ns_ node_ rtProtos_ rtVia_ nextHop_ rtpref_ metric_
    foreach p [array names rtProtos_] {
        $rtProtos_($p) intf-changed
        $rtProtos_($p) compute-routes
    }
    $self compute-routes
}

rtObject instproc dump-routes chan {
    $self instvar ns_ node_ nextHop_ rtpref_ metric_ rtVia_

#   if {[info proc SplitObjectCompare] == ""} {
#       # XXX: in tcl8 we need to find this in the global namespace
#       if {[info proc ::SplitObjectCompare] == {}} {
#           puts stderr "${class}::${proc} failed.  Update your TclCL library"
#           return
#       }
#   }

    if {$ns_ != ""} {
        set time [$ns_ now]
    } else {
        set time 0.0
    }
    puts $chan [concat "Node:\t${node_}([$node_ id])\tat t ="            \
            [format "%4.2f" $time]]
    puts $chan "  Dest\t\t nextHop\tPref\tMetric\tProto"
    foreach dest [lsort -command SplitObjectCompare [$ns_ all-nodes-list]] {
        if {[llength $nextHop_($dest)] > 1} {
            set p [split [$rtVia_($dest) info class] /]
            set proto [lindex $p [expr [llength $p] - 1]]
            foreach rt $nextHop_($dest) {
                puts $chan [format "%-5s(%d)\t%-5s(%d)\t%3d\t%4d\t %s"      \
                        $dest [$dest id] $rt [[$rt set toNode_] id]          \
                        $rtpref_($dest) $metric_($dest) $proto]
            }
        } elseif {$nextHop_($dest) != ""} {
            set p [split [$rtVia_($dest) info class] /]
            set proto [lindex $p [expr [llength $p] - 1]]
```

```
            puts $chan [format "%-5s(%d)\t%-5s(%d)\t%3d\t%4d\t %s"    \
                    $dest [$dest id]                                  \
                    $nextHop_($dest) [[$nextHop_($dest) set toNode_] id] \
                    $rtpref_($dest) $metric_($dest) $proto]
        } elseif {$dest == $node_} {
            puts $chan [format "%-5s(%d)\t%-5s(%d)\t%03d\t%4d\t %s"  \
                    $dest [$dest id] $dest [$dest id] 0 0 "Local"]
        } else {
            puts $chan [format "%-5s(%d)\t%-5s(%s)\t%03d\t%4d\t %s"  \
                    $dest [$dest id] "" "-" 255 32 "Unknown"]
        }
    }
}

rtObject instproc rtProto? proto {
    $self instvar rtProtos_
    if [info exists rtProtos_($proto)] {
        return $rtProtos_($proto)
    } else {
        return ""
    }
}


rtObject instproc nextHop? dest {
    $self instvar nextHop_
    $self set nextHop_($dest)
}

rtObject instproc rtpref? dest {
    $self instvar rtpref_
    $self set rtpref_($dest)
}

rtObject instproc metric? dest {
    $self instvar metric_
    $self set metric_($dest)
}

#_
Class rtPeer

rtPeer instproc init {addr cls} {
    $self next
    $self instvar addr_ metric_ rtpref_
    set addr_ $addr
    foreach dest [[Simulator instance] all-nodes-list] {
        set metric_($dest) [$cls set INFINITY]
        set rtpref_($dest) [$cls set preference_]
    }
```

```
}

rtPeer instproc addr? {} {
   $self instvar addr_
   return $addr_
}

rtPeer instproc metric {dest val} {
   $self instvar metric_
   set metric_($dest) $val
}

rtPeer instproc metric? dest {
   $self instvar metric_
   return $metric_($dest)
}

rtPeer instproc preference {dest val} {
   $self instvar rtpref_
   set rtpref_($dest) $val
}

rtPeer instproc preference? dest {
   $self instvar rtpref_
   return $rtpref_($dest)
}

#_
#Class Agent/rtProto -superclass Agent

Agent/rtProto proc pre-init-all args {
   # By default, do nothing when a person does $ns rtproto foo.
}

Agent/rtProto proc init-all args {
   error "No initialization defined"
}

Agent/rtProto instproc init node {
   $self next

   $self instvar ns_ node_ rtObject_ preference_ ifs_ ifstat_
   set ns_ [Simulator instance]

   catch "set preference_ [[$self info class] set preference_]" ret
   if { $ret == "" } {
         set preference_ [$class set preference_]
   }
   foreach nbr [$node set neighbor_] {
         set link [$ns_ link $node $nbr]
```

```
        set ifs_($nbr) $link
        set ifstat_($nbr) [$link up?]
    }
    set rtObject_ [$node rtObject?]
}

Agent/rtProto instproc compute-routes {} {
    error "No route computation defined"
}

Agent/rtProto instproc intf-changed {} {
    #NOTHING
}

Agent/rtProto instproc send-updates args {
    #NOTHING
}

Agent/rtProto proc compute-all {} {
    #NOTHING
}

#
# Static routing, the default
#
Class Agent/rtProto/Static -superclass Agent/rtProto

Agent/rtProto/Static proc init-all args {
    # The Simulator knows the entire topology.
    # Hence, the current compute-routes method in the Simulator class is
    # well suited.  We use it as is.

    [Simulator instance] compute-routes

}

#
# Session based unicast routing
#
Class Agent/rtProto/Session -superclass Agent/rtProto

Agent/rtProto/Session proc init-all args {
    [Simulator instance] compute-routes
}

Agent/rtProto/Session proc compute-all {} {
    [Simulator instance] compute-routes

}
```

```
#_
#############################################################################
#
# Code below this line is experimental, and should be considered work
# in progress.  None of this code is used in production test-suites, or
# in the release yet, and hence should not be a problem to anyone.
#
Class Agent/rtProto/Direct -superclass Agent/rtProto
Agent/rtProto/Direct instproc init node {
   $self next $node
   $self instvar ns_ rtpref_ nextHop_ metric_ ifs_

   foreach node [$ns_ all-nodes-list] {
         set rtpref_($node) 255
         set nextHop_($node) ""
         set metric_($node) -1
   }
   foreach node [array names ifs_] {
         set rtpref_($node) [$class set preference_]
   }
}

Agent/rtProto/Direct instproc compute-routes {} {
   $self instvar ifs_ ifstat_ nextHop_ metric_ rtsChanged_
   set rtsChanged_ 0
   foreach nbr [array names ifs_] {
         if {$nextHop_($nbr) == "" && [$ifs_($nbr) up?] == "up"} {
            set ifstat_($nbr) 1
            set nextHop_($nbr) $ifs_($nbr)
            set metric_($nbr) [$ifs_($nbr) cost?]
            incr rtsChanged_
         } elseif {$nextHop_($nbr) != "" && [$ifs_($nbr) up?] != "up"} {
            set ifstat_($nbr) 0
            set nextHop_($nbr) ""
            set metric_($nbr) -1
            incr rtsChanged_
         }
   }
}
#
# Distance Vector Route Computation
#
# Class Agent/rtProto/DV -superclass Agent/rtProto
Agent/rtProto/DV set UNREACHABLE[rtObject set unreach_]
Agent/rtProto/DV set mid_                0

# added for rtProtoSAAM
# Class Agent/rtProto/SAAM -superclass Agent/rtProto
Agent/rtProto/SAAM set UNREACHABLE      [rtObject set unreach_]
Agent/rtProto/SAAM set mid_              0
```

```
Agent/rtProto/DV proc init-all args {
  if { [llength $args] == 0 } {
        set nodeslist [[Simulator instance] all-nodes-list]
  } else {
        eval "set nodeslist $args"
  }
  Agent set-maxttl Agent/rtProto/DV INFINITY
  eval rtObject init-all $nodeslist
  foreach node $nodeslist {
        set proto($node) [[$node rtObject?] add-proto DV $node]
  }
  foreach node $nodeslist {
        foreach nbr [$node neighbors] {
          set rtobj [$nbr rtObject?]
          if { $rtobj != "" } {
                set rtproto [$rtobj rtProto? DV]
                if { $rtproto != "" } {
                  $proto($node) add-peer $nbr [$rtproto set addr_]
                }
          }
        }
  }
}

# added for rtProtoSAAM
Agent/rtProto/SAAM proc init-all args {
  if { [llength $args] == 0 } {
        set nodeslist [[Simulator instance] all-nodes-list]
  } else {
        eval "set nodeslist $args"
  }
  Agent set-maxttl Agent/rtProto/SAAM INFINITY
  eval rtObject init-all $nodeslist
  foreach node $nodeslist {
        set proto($node) [[$node rtObject?] add-proto SAAM $node]
  }
  foreach node $nodeslist {
        foreach nbr [$node neighbors] {
          set rtobj [$nbr rtObject?]
          if { $rtobj != "" } {
                set rtproto [$rtobj rtProto? SAAM]
                if { $rtproto != "" } {
                  $proto($node) add-peer $nbr [$rtproto set addr_]
                }
          }
        }
  }

}
```

```
Agent/rtProto/DV instproc init node {
   global rtglibRNG

   $self next $node
   $self instvar ns_ rtObject_ ifsUp_
   $self instvar preference_ rtpref_ nextHop_ nextHopPeer_ metric_ multiPath_

   set UNREACHABLE [$class set UNREACHABLE]
   foreach dest [$ns_ all-nodes-list] {
         set rtpref_($dest) $preference_
         set nextHop_($dest) ""
         set nextHopPeer_($dest) ""
         set metric_($dest)  $UNREACHABLE
   }
   set ifsUp_ ""
   set multiPath_ [[$rtObject_ set node_] set multiPath_]
   set updateTime [$rtglibRNG uniform 0.0 0.5]
   $ns_ at $updateTime "$self send-periodic-update"
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc init node {
   global rtglibRNG

   $self next $node
   $self instvar ns_ rtObject_ ifsUp_
   $self instvar preference_ rtpref_ nextHop_ nextHopPeer_ metric_ multiPath_

   set UNREACHABLE [$class set UNREACHABLE]
   foreach dest [$ns_ all-nodes-list] {
         set rtpref_($dest) $preference_
         set nextHop_($dest) ""
         set nextHopPeer_($dest) ""
         set metric_($dest)  $UNREACHABLE
   }
   set ifsUp_ ""
   set multiPath_ [[$rtObject_ set node_] set multiPath_]
   set updateTime [$rtglibRNG uniform 0.0 0.5]
   $ns_ at $updateTime "$self send-periodic-update"
}

Agent/rtProto/DV instproc add-peer {nbr agentAddr} {
   $self instvar peers_
   $self set peers_($nbr) [new rtPeer $agentAddr $class]
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc add-peer {nbr agentAddr} {
   $self instvar peers_
   $self set peers_($nbr) [new rtPeer $agentAddr $class]
```

```
}

Agent/rtProto/DV instproc send-periodic-update {} {
  global rtglibRNG

  $self instvar ns_
  $self send-updates 1  ;# Anything but 0
  set updateTime [expr [$ns_ now] + \
        ([$class set advertInterval] * [$rtglibRNG uniform 0.9 1.1])]
  $ns_ at $updateTime "$self send-periodic-update"
}

#added for rtProtoSAAM
Agent/rtProto/SAAM instproc send-periodic-update {} {
  global rtglibRNG

  $self instvar ns_
  $self send-updates 1  ;# Anything but 0
  set updateTime [expr [$ns_ now] + \
        ([$class set advertInterval] * [$rtglibRNG uniform 0.9 1.1])]
  $ns_ at $updateTime "$self send-periodic-update"
}

Agent/rtProto/DV instproc compute-routes {} {
  $self instvar ns_ ifs_ rtpref_ metric_ nextHop_ nextHopPeer_
  $self instvar peers_ rtsChanged_ multiPath_

  set INFINITY [$class set INFINITY]
  set MAXPREF  [rtObject set maxpref_]
  set UNREACH        [rtObject set unreach_]
  set rtsChanged_ 0
  foreach dst [$ns_ all-nodes-list] {
      set p [lindex $nextHopPeer_($dst) 0]
      if {$p != ""} {
        set metric_($dst) [$p metric? $dst]
        set rtpref_($dst) [$p preference? $dst]
      }

      set pf $MAXPREF
      set mt $INFINITY
      set nh(0) 0
      foreach nbr [array names peers_] {
        set pmt [$peers_($nbr) metric? $dst]
        set ppf [$peers_($nbr) preference? $dst]

        # if peer metric not valid      continue
        # if peer pref higher           continue
        # if peer pref lower            set to latest values
        # else peer pref equal
```

```
    #       if peer metric higher     continue
    #       if peer metric lower      set to latest values
    #       else peer metrics equal   append latest values

    if { $pmt < 0 || $pmt >= $INFINITY || $ppf > $pf || $pmt > $mt } \
            continue
    if { $ppf < $pf || $pmt < $mt } {
            set pf $ppf
            set mt $pmt
            unset nh            ;# because we must compute *new* next hops
    }
    set nh($ifs_($nbr)) $peers_($nbr)
}
catch "unset nh(0)"
if { $pf == $MAXPREF && $mt == $INFINITY } continue
if { $pf > $rtpref_($dst) ||                                    \
            ($metric_($dst) >= 0 && $mt > $metric_($dst)) }      \
            continue
if {$mt >= $INFINITY} {
    set mt $UNREACH
}

incr rtsChanged_
if { $pf < $rtpref_($dst) || $mt < $metric_($dst) } {
    set rtpref_($dst) $pf
    set metric_($dst) $mt
    set nextHop_($dst) ""
    set nextHopPeer_($dst) ""
    foreach n [array names nh] {
            lappend nextHop_($dst) $n
            lappend nextHopPeer_($dst) $nh($n)
            if !$multiPath_ break;
    }
    continue
}

set rtpref_($dst) $pf
set metric_($dst) $mt
set newNextHop ""
set newNextHopPeer ""
foreach rt $nextHop_($dst) {
    if [info exists nh($rt)] {
            lappend newNextHop $rt
            lappend newNextHopPeer $nh($rt)
            unset nh($rt)
    }
}
set nextHop_($dst) $newNextHop
set nextHopPeer_($dst) $newNextHopPeer
if { $multiPath_ || $nextHop_($dst) == "" } {
```

```
            foreach rt [array names nh] {
                lappend nextHop_($dst) $rt
                lappend nextHopPeer_($dst) $nh($rt)
                if !$multiPath_ break
            }
        }
    }
    set rtsChanged_
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc compute-routes {} {
    $self instvar ns_ ifs_ rtpref_ metric_ nextHop_ nextHopPeer_
    $self instvar peers_ rtsChanged_ multiPath_

    set INFINITY [$class set INFINITY]
    set MAXPREF  [rtObject set maxpref_]
    set UNREACH          [rtObject set unreach_]
    set rtsChanged_ 0
    foreach dst [$ns_ all-nodes-list] {
        set p [lindex $nextHopPeer_($dst) 0]
        if {$p != ""} {
            set metric_($dst) [$p metric? $dst]
            set rtpref_($dst) [$p preference? $dst]
        }

        set pf $MAXPREF
        set mt $INFINITY
        set nh(0) 0
        foreach nbr [array names peers_] {
            set pmt [$peers_($nbr) metric? $dst]
            set ppf [$peers_($nbr) preference? $dst]

            # if peer metric not valid      continue
            # if peer pref higher           continue
            # if peer pref lower            set to latest values
            # else peer pref equal
            #     if peer metric higher     continue
            #     if peer metric lower      set to latest values
            #     else peer metrics equal   append latest values

            if { $pmt < 0 || $pmt >= $INFINITY || $ppf > $pf || $pmt > $mt } \
                    continue
            if { $ppf < $pf || $pmt < $mt } {
                set pf $ppf
                set mt $pmt
                unset nh          ;# because we must compute *new* next hops
            }
            set nh($ifs_($nbr)) $peers_($nbr)
        }
```

```
        catch "unset nh(0)"
        if { $pf == $MAXPREF && $mt == $INFINITY } continue
        if { $pf > $rtpref_($dst) ||                                    \
                ($metric_($dst) >= 0 && $mt > $metric_($dst)) }          \
                continue
        if {$mt >= $INFINITY} {
          set mt $UNREACH
        }

        incr rtsChanged_
        if { $pf < $rtpref_($dst) || $mt < $metric_($dst) } {
          set rtpref_($dst) $pf
          set metric_($dst) $mt
          set nextHop_($dst) ""
          set nextHopPeer_($dst) ""
          foreach n [array names nh] {
                lappend nextHop_($dst) $n
                lappend nextHopPeer_($dst) $nh($n)
                if !$multiPath_ break;
          }
          continue
        }

        set rtpref_($dst) $pf
        set metric_($dst) $mt
        set newNextHop ""
        set newNextHopPeer ""
        foreach rt $nextHop_($dst) {
          if [info exists nh($rt)] {
                lappend newNextHop $rt
                lappend newNextHopPeer $nh($rt)
                unset nh($rt)
          }
        }
        set nextHop_($dst) $newNextHop
        set nextHopPeer_($dst) $newNextHopPeer
        if { $multiPath_ || $nextHop_($dst) == "" } {
          foreach rt [array names nh] {
                lappend nextHop_($dst) $rt
                lappend nextHopPeer_($dst) $nh($rt)
                if !$multiPath_ break
          }
        }
  }
  set rtsChanged_
}


Agent/rtProto/DV instproc intf-changed {} {
  $self instvar ns_ peers_ ifs_ ifstat_ ifsUp_ nextHop_ nextHopPeer_ metric_
```

```
    set INFINITY [$class set INFINITY]
    set ifsUp_ ""
    foreach nbr [array names peers_] {
        set state [$ifs_($nbr) up?]
        if {$state != $ifstat_($nbr)} {
            set ifstat_($nbr) $state
            if {$state != "up"} {
                if ![info exists all-nodes] {
                    set all-nodes [$ns_ all-nodes-list]
                }
                foreach dest ${all-nodes} {
                    $peers_($nbr) metric $dest $INFINITY
                }
            } else {
                lappend ifsUp_ $nbr
            }
        }
    }
}

#added for rtProtoSAAM
Agent/rtProto/SAAM instproc intf-changed {} {
    $self instvar ns_ peers_ ifs_ ifstat_ ifsUp_ nextHop_ nextHopPeer_ metric_
    set INFINITY [$class set INFINITY]
    set ifsUp_ ""
    foreach nbr [array names peers_] {
        set state [$ifs_($nbr) up?]
        if {$state != $ifstat_($nbr)} {
            set ifstat_($nbr) $state
            if {$state != "up"} {
                if ![info exists all-nodes] {
                    set all-nodes [$ns_ all-nodes-list]
                }
                foreach dest ${all-nodes} {
                    $peers_($nbr) metric $dest $INFINITY
                }
            } else {
                lappend ifsUp_ $nbr
            }
        }
    }
}

Agent/rtProto/DV proc get-next-mid {} {
    set ret [Agent/rtProto/DV set mid_]
    Agent/rtProto/DV set mid_ [expr $ret + 1]
    set ret
}

# added for rtProtoSAAM
```

```
Agent/rtProto/SAAM proc get-next-mid {} {
   set ret [Agent/rtProto/SAAM set mid_]
   Agent/rtProto/SAAM set mid_ [expr $ret + 1]
   set ret
}

Agent/rtProto/DV proc retrieve-msg id {
   set ret [Agent/rtProto/DV set msg_($id)]
   Agent/rtProto/DV unset msg_($id)
   set ret
}

# added for rtProtoSAAM
Agent/rtProto/SAAM proc retrieve-msg id {
   set ret [Agent/rtProto/SAAM set msg_($id)]
   Agent/rtProto/SAAM unset msg_($id)
   set ret
}

Agent/rtProto/DV instproc send-updates changes {
   $self instvar peers_ ifs_ ifsUp_

   if $changes {
         set to-send-to [array names peers_]
   } else {
         set to-send-to $ifsUp_
   }
   set ifsUp_ ""
   foreach nbr ${to-send-to} {
         if { [$ifs_($nbr) up?] == "up" } {
            $self send-to-peer $nbr
         }
   }
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc send-updates changes {
   $self instvar peers_ ifs_ ifsUp_

   if $changes {
         set to-send-to [array names peers_]
   } else {
         set to-send-to $ifsUp_
   }
   set ifsUp_ ""
   foreach nbr ${to-send-to} {
         if { [$ifs_($nbr) up?] == "up" } {
            $self send-to-peer $nbr
         }
   }
```

```
}

Agent/rtProto/DV instproc send-to-peer nbr {
   $self instvar ns_ rtObject_ ifs_ peers_
   set INFINITY [$class set INFINITY]
   foreach dest [$ns_ all-nodes-list] {
         set metric [$rtObject_ metric? $dest]
         if {$metric < 0} {
            set update($dest) $INFINITY
         } else {
            set update($dest) [$rtObject_ metric? $dest]
            foreach nh [$rtObject_ nextHop? $dest] {
                  if {$nh == $ifs_($nbr)} {
                     set update($dest) $INFINITY
                  }
            }
         }
   }
   set id [$class get-next-mid]
   $class set msg_($id) [array get update]
#   set n [$rtObject_ set node_];
   puts stderr [concat [format ">>> %7.5f" [$ns_ now]]
            "${n}([$n id]/[$self set addr_]) send update"
            "to ${nbr}([$nbr id]/[$peers_($nbr) addr?]) id = $id"];
   set j 0;
   foreach i [lsort -command TclObjectCompare [array names update]] {
         puts -nonewline "\t${i}([$i id]) $update($i)";
      if {$j == 3} {
            puts "";
         };
         set j [expr ($j + 1) % 4];
   };
   if $j { puts ""; }

   # XXX Note the singularity below...
   $self send-update [$peers_($nbr) addr?] $id [array size update]
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc send-to-peer nbr {
   $self instvar ns_ rtObject_ ifs_ peers_
   set INFINITY [$class set INFINITY]
   foreach dest [$ns_ all-nodes-list] {
         set metric [$rtObject_ metric? $dest]
         if {$metric < 0} {
            set update($dest) $INFINITY
         } else {
            set update($dest) [$rtObject_ metric? $dest]
            foreach nh [$rtObject_ nextHop? $dest] {
                  if {$nh == $ifs_($nbr)} {
```

```
                set update($dest) $INFINITY
            }
        }
    }
    set id [$class get-next-mid]
    $class set msg_($id) [array get update]
#   set n [$rtObject_ set node_];
    puts stderr [concat [format ">>> %7.5f" [$ns_ now]]
            "${n}([$n id]/[$self set addr_]) send update"
            "to ${nbr}([$nbr id]/[$peers_($nbr) addr?]) id = $id"];
    set j 0;
    foreach i [lsort -command TclObjectCompare [array names update]] {
        puts -nonewline "\t${i}([$i id]) $update($i)";
      if {$j == 3} {
            puts "";
        };
        set j [expr ($j + 1) % 4];
    };
    if $j { puts ""; }

    # XXX Note the singularity below...
    $self send-update [$peers_($nbr) addr?] $id [array size update]
}
Agent/rtProto/DV instproc recv-update {peerAddr id} {
    $self instvar peers_ ifs_ nextHopPeer_ metric_
    $self instvar rtsChanged_ rtObject_

    set INFINITY [$class set INFINITY]
    set UNREACHABLE  [$class set UNREACHABLE]
    set msg [$class retrieve-msg $id]
    array set metrics $msg
#   set n [$rtObject_ set node_];                                   \
    puts stderr [concat [format "<<< %7.5f" [[Simulator instance] now]]   \
            "${n}([$n id]) recv update from peer $peerAddr id = $id"]
    foreach nbr [array names peers_] {
        if {[$peers_($nbr) addr?] == $peerAddr} {
            set peer $peers_($nbr)
            if { [array size metrics] > [Node set nn_] } {
                error "$class::$proc update $peerAddr:$msg:$count is larger than the simulation
topology"
            }
            set metricsChanged 0
            foreach dest [array names metrics] {
            set metric [expr $metrics($dest) + [$ifs_($nbr) cost?]]
                if {$metric > $INFINITY} {
                    set metric $INFINITY
                }
                if {$metric != [$peer metric? $dest]} {
                    $peer metric $dest $metric
```

```
                incr metricsChanged
            }
        }
        if $metricsChanged {
            $self compute-routes
            incr rtsChanged_ $metricsChanged
            $rtObject_ compute-routes
        } else {
            # dynamicDM multicast hack.
            # If we get a message from a neighbour, then something
            # at that neighbour has changed.  While this may not
            # cause any unicast changes on our end, dynamicDM
            # looks at neighbour's routing tables to compute
            # parent-child relationships, and has to do them
            # again.
            #
            $rtObject_ flag-multicast -1
        }
        return
    }
  }
  error "$class::$proc update $peerAddr:$msg:$count from unknown peer"
}

# added for rtProtoSAAM
Agent/rtProto/SAAM instproc recv-update {peerAddr id} {
  $self instvar peers_ ifs_ nextHopPeer_ metric_
  $self instvar rtsChanged_ rtObject_

  set INFINITY [$class set INFINITY]
  set UNREACHABLE  [$class set UNREACHABLE]
  set msg [$class retrieve-msg $id]
  array set metrics $msg
#   set n [$rtObject_ set node_];                                    \
  puts stderr [concat [format "<<< %7.5f" [[Simulator instance] now]]  \
          "${n}([$n id]) recv update from peer $peerAddr id = $id"]
  foreach nbr [array names peers_] {
      if {[$peers_($nbr) addr?] == $peerAddr} {
          set peer $peers_($nbr)
          if { [array size metrics] > [Node set nn_] } {
              error "$class::$proc update $peerAddr:$msg:$count is larger than the simulation
topology"
          }
          set metricsChanged 0
          foreach dest [array names metrics] {
          set metric [expr $metrics($dest) + [$ifs_($nbr) cost?]]
              if {$metric > $INFINITY} {
                  set metric $INFINITY
              }
              if {$metric != [$peer metric? $dest]} {
```

```
                $peer metric $dest $metric
                incr metricsChanged
            }
        }
        if $metricsChanged {
            $self compute-routes
            incr rtsChanged_ $metricsChanged
            $rtObject_ compute-routes
        } else {
            # dynamicDM multicast hack.
            # If we get a message from a neighbour, then something
            # at that neighbour has changed.  While this may not
            # cause any unicast changes on our end, dynamicDM
            # looks at neighbour's routing tables to compute
            # parent-child relationships, and has to do them
            # again.
            #
            $rtObject_ flag-multicast -1
        }
        return
    }
  }
  error "$class::$proc update $peerAddr:$msg:$count from unknown peer"
}
Agent/rtProto/DV proc compute-all {} {
  # Because proc methods are not inherited from the parent class.
}

# added for rtProtoSAAM
Agent/rtProto/SAAM proc compute-all {} {
  # Because proc methods are not inherited from the parent class.
}

#
# Manual routing
#
Class Agent/rtProto/Manual -superclass Agent/rtProto

Agent/rtProto/Manual proc pre-init-all args {
  Simulator set node_factory_ ManualRtNode
}

Agent/rtProto/Manual proc init-all args {
  # The user will do all routing.
}
### Local Variables:
### mode: tcl
### tcl-indent-level: 4
### tcl-default-application: ns
### End:
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[FALL99] Kevin Fall and Kannan Varadhan, *NS Notes and Documentation,* The VINT Project, 1999.

[KESH88] Srinivasan Keshav, *REAL: A Network Simulator,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1988.

[NATI97] National Coordination Office for Computing, Information, and Communications, *Next Generation Internet Initiative,* http://www.ccic.gov/ngi/concept-Jul97/, (HTML Document), 1997.

[SCHO96] Herbert Schorr and Deborah Estrin, *Virtual InterNetwork Testbed(VINT): methods and system*, Information Sciences Institute, University of Southern California, Los Angeles, 1996.

[XIE98] Geoff Xie, *SAAM: Network Management for Integrated Services*, Department of Computer Science, Naval Postgraduate School, Monterey CA, 1998.

[GUER98] Guerin, *QoS Routing Mechanisms and OSPF Extensions*, Internet Engineering Task Force, 1998.

[VRAB99] Dean Vrable and Jon Yarger, *The SAAM Architecture: Enabling Integrated Services*, Department of Computer Science, Naval Postgraduate School, Monterey CA, 1999

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..... 2
   28725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library..... 2
   Naval Postgraduate School
   411 Dyer Rd. Monterey, California 93940-5101

3. Director, Training and Education..... 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, Virginia 22134-5027

4. Director, Marine Corps Research Center..... 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, Virginia 22134-5107

5. Director, Studies and Analysis Division..... 1
   MCCDC, Code C45
   300 Russell Road
   Quantico, Virginia 22134-5130

6. Marine Corps Representative..... 1
   Naval Postgraduate School
   Code 037, Bldg. 330, IN-116
   555 Dyer Road
   Monterey, CA 93940

7. Chairman,Code CS..... 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

8. Dr. Geoffrey Xie..... 1
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

9. CDR Michael J. Holden.....1
Computer Science Department, Code CS/HM
Naval Postgraduate School
Monterey, California 93943-5100

10. Maj Brian Tiefert.....1
2205 Star Lane
Albany, GA 31707